Queen Mary
University of London

Sensitivity computation and shape optimisation in aerodynamics using the adjoint methodology and Automatic Differentiation

Faidon Christakopoulos
School of Engineering and Materials Science
Queen Mary
University of London

Thesis submitted for the degree of Doctor of Philosophy

Supervisor :
Dr. Jens - Dominik Müller

9 September 2012

Abstract

Adjoint based optimisation has until now demonstrated a great promise for optimisation in aerodynamics due to its independence of the number of design variables. This is essential in large industrial applications, where hundreds of parameters might be needed so as to describe the geometry. Although the computational cost of the methodology is smaller than that of stochastic optimisation methods, the implementation and related program maintenance time and effort could be particularly high.

The aim of the present is to contribute to the effort of redusing the cost above by examining whether programs using the adjoint methodology for optimisation can be automatically generated and maintained via Automatic Differentiation, while presenting comparable performance to hand derived adjoints. This could lead to accurate adjoint based optimisation codes, which would inherit any change or addition to the relative original Computational Fluid Dynamics code.

Such a methodology is presented and all the different steps involved are detailed. It is found that although a considerable initial effort is required for preparation of the source code for differentiation, hand assembly of the sensitivity algorithms and scripting for the automation of the entire process, the target of this research program is achieved and fully automatically generated adjoint codes with comparable performance can be acquired. After applying the methodology to a number of aerodynamic shape optimisation examples, the logic is also extended to higher derivatives, which could also be included in the optimisation process for robust design.

Thank you note

Studies towards a PhD are a demanding and difficult path, with many ups and downs. It is a journey through joy, anxiety, disappointment, hope and pride, feelings that circle for its entire duration. This journey is very enjoyable though, when there are special people around you to share the joy, to support and to give you hope. I have been very lucky to have had many unique people around me all these years and this thesis is dedicated to them. This letter is just a small thank you to you all.

I would like to express my gratitude to my supervisor, Dr Jens-Dominic Müller, for giving me the opportunity to pursue this research, for his continuous support and mentoring and for the extensive amount of knowledge he has transmitted to me gained through these years. It has been an honour and a privilege to be a student of his. Thank you very much.

This would all be impossible without family, Gianni, Evi and Ismini. Thank you for being the best family I could ever ask for, for supporting me in every step in my life and for giving me the motivation to chase my dreams.

Through my PhD years, I was fortunate to meet a unique person that has walked every step with me in this journey. Inci a thank you is not enough for being there with and for me. This also includes the people that make me feel like having a second family. Tesekkurler Osman, Hatice, Ipek and Ilker.

A thank you goes to all my friends in the office for their company and continuous support. Neveen, Henry, Milan, Percy, Shenren, Mateusz and Andrei thank you all very much. A special thank you goes to my good friend Stephan Hunkeler, for the numerous hours in and out of the office and the unlimited support.

Closing this letter, I would like to thank my examiners, Prof. Bruce Christianson and Dr Stephan Schmidt, for having the interest and taking the time to read the thesis and perform the viva, as well as for their useful comments.

Faidon

Contents

List of Figures

List of Tables

List of Symbols

A	Jacobian matrix
c	speed of sound
C_D	drag coefficient
CFL	Courant-Friedruch-Lewy number
C_L	lift coefficient
C_P	pressure coefficient
f	Tangent source term
H	total enthalpy
$\vec{F_c}$	vector of convective fluxes
$\vec{F_v}$	vector of viscous fluxes
J	cost function
k	thermal conductivity coefficient
Ma	Mach number
$\vec{n}$	unit normal vector of control volume face
$\vec{n_x}, \vec{n_y}, \vec{n_z}$	components of unit normal vector
P	pressure
R	Residuals
T	temperature
Δt	time step
$\vec{U}$	vector of conservative variables ($[\rho, \rho u, \rho v, \rho w, \rho E]^T$)
u, v, w	Cartesian velocity components
v	adjoint variables
V	contravariant velocity
Δx	cell size in X-direction
Λ_c	eigenvalue of convective flux Jacobian
$\Lambda_{c,p}$	eigenvalue of convective flux Jacobian (precondtioned)

ρ	density
τ	viscous stress
Ω	control volume
Φ	Continuous function
E	Energy
Re	Reynolds number
Pr	Prandtl number
$\vec{\tau}$	stress tensor
$\vec{F}$	Vector of body forces
$\vec{q}$	Heat flux vector
μ	Dynamic viscosity
ν	Kinematic viscosity
a	Thermal diffusivity
e	Internal energy per unit mass
k	Thermal conductivity
$\vec{U}_{L,R}$	Vector of left and right state
$\mathbf{A}_{Roe}$	Roe's Jacobian matrix
Ψ	Flux limiter
P_{tar}	Target pressure
I_H^h	Prolongation operator
$\mathcal{B}$	Block Jacobi pre–conditioner
ψ	Field to be smoothed
σ	Design scaling factor
ξ	Global scaling factor
g_{RMS}	Stopping criterion for the one–shot methodology
ToT	Tangent over Tangent
ToR	Tangent over Reverse
RoT	Reverse over Tangent
RoR	Reverse over Reverse

Chapter 1

Introduction

1.1 Evolution in fluid mechanics

Fluids cover the greater percentage of the earth and have therefore always played an important role in the lives of people and animals. Whether these are air, water or anything else of similar type, fluids surround every aspect of life as we know it. As a result, it was inevitable for the mankind to experiment with fluids and look for ways to ease life.

In the pursuit to understand the laws of fluid flow, experimental facilities were built. Throughout the years, there has been a lot of progress in this area, resulting in modern wind tunnels (Figure 1.1) and experimental hydrodynamics channels, equipped with high technology measurement techniques. Tests carried out in those provide real life results with very high accuracy.

The experimental process though has always proved to be expensive. Considering the variety of applications that include fluid flow, especially regarding engineering products, the building, maintenance and use of such facilities soon becomes financially expensive. With every engineering idea, a model would have to be constructed, tested, evaluated, adjusted, rebuilt, retested and so on. Such a procedure is time consuming and can be prohibitive for engineers with good ideas but insufficient funds. Even for large corporations, such as those in the aeronautical industry, these costs needed to be reduced.

For this reason, scientists and engineers began exploring the physics behind fluid flows and trying to describe them with physical models. Along with the evolution in computer science, this gave birth to a new discipline for fluid mechanics in the mid to late 20^{th} century : the *Computational Fluid Dynamics* (CFD). The effort of this new discipline was to numerically solve the equations that describe fluid flows on computers by transforming the physical fluid models into computer programs. In the beginning, such computations were of low accuracy and high computational cost. With the years

Figure 1.1: Modern experimental wind tunnel at NASA research center (photo courtesy of NASA, www.nasa.gov).

this has changed though and very accurate numerical models are available nowadays, which have been extensively validated through experiments versus computations. These can be used either on powerful personal computers or supercomputers to compute the flow solutions in just a few minutes, for small problems, or a few hours, for bigger ones. Today, most of the industry has replaced the largest part of their former experimental design process with high fidelity CFD packages, reducing the development time and cost. Only in the final design stage are experiments carried out and that is, if the designers choose to validate the results from CFD.

Despite the fact that Computational Fluid Dynamics became an established and widely used discipline, the design process of aero or hydrodynamic shapes still lacks automation. Such design was still based on experience and a series of trial and error experiments; not experimental but numerical experiments this time. So, in order to improve the performance of the designed components there is still need of setting up the numerical problem, evaluating the results, reforming the shape, retesting etc.. An additional factor to this was that, the whole process heavily relies on the experience of the designers and was built throughout many years. And even with practical experience, this does not imply that the final results will be optimal. As a result, there is still need for automatic optimisation of the design.

In most recent years, this problem was dealt with by using genetic algorithms. These are based on stochastic methodologies and rules inspired from the evolution theory of Darwin. Although this is a powerful methodology, it is computationally expensive, as it

requires the solution of the complicated flow equations many times, in order to generate "populations" and apply the rules of evolution. Therefore, although it does provide optimal shapes automatically, its cost can still be prohibitive, due to the runtime and computational costs.

On the other hand, gradient based optimisation methods use the derivative information to drive a problem – in this case a shape – fast to its nearest extremal point, a minimum or a maximum. Such methods though require the derivative of a cost function with respect to all design variables and (for those not following the adjoint methodology) their cost can soon become prohibitive for large real world optimisation cases with a large number of design variables. This problem is eliminated with the use of the adjoint methodology, which is able to compute the sensitivity information independent of the number of design variables. This was used in aerodynamic shape optimisation by Jameson [51] as well as others [91]. The methodology was used through the years that followed, but although it is a fast optimisation method, the derivation and implementation of the sensitivity systems to solve was complex and time consuming. This is the reason why the idea was not widely adopted from the beginning. It is characteristic to say that only recently have adjoint systems appeared, which are derived from the complete Navier–Stokes equations. But the latest years also gave rise to the growing discipline of algorithmic differentiation. Using this methodology (Appendix A, sensitivities of functions can be computed by differentiating the algorithm that computes them. this gave birth to Automatic Differentiation (AD) tools, which perform this operation. Despite what the name might imply though, the use and application of such tools is not completely automatic and still presents many difficulties. Therefore, a fully automated adjoint based optimisation methodology is still missing.

This is where the present thesis contributes to the field, by delivering a fully automated adjoint based optimisation methodology, with the aid of AD. In the chapters to follow, the ingredients involved are going to be presented. Also, the present research moved even further by the examination of automated second derivative code generation and computation of Hessians via AD. Apart for the novel methodologies for the complete automatic derivation of first and second order sensitivity systems, issues of performance acceleration are examined and a novel methodology for the acceleration of second order systems' convergence is presented. All the above are coupled with other performance improvement techniques, such as multi–grid and pre–conditioning.

The following sections in this chapter discuss the problem of optimisation and give a general overview before the work of this thesis is presented.

1.2 Optimisation methodologies

Optimisation essentially is to find the location of a local or global extremum of a given function. This could be a simple mathematical function with just a single independent parameter, the complex Navier–Stokes flow equations or any other type of function. Since the beginning the optimisation concept in mathematics and science, many optimisation strategies and algorithms have been developed. In fact, nowadays their number is so high, that a whole book would needed to describe them. For that reason, only the most important methods commonly used in the area of aerodynamics are going to be dealt with in the present thesis.

There are two main families of optimisation methods used in aerodynamics, that follow different approaches in finding the optimum. Following the methodology used, they could be described as *gradient*–based or *stochastic* based optimisation. The category of stochastic–based optimisation includes the popular method of genetic algorithms, that consists the main body of the category as well. The logic is based on stochastic methods and frequently on the evolutionary theory of Darwin. This logic is used in aerodynamic shape optimisation but has proven to be very time consuming and computationally expensive, since it involves a high number of flow evaluations, which drive the design to the optimum. Such a process can indeed lead to the global minimum of any function but it is so time consuming that its cost soon becomes prohibitive for large industrial applications. Apart from this, it should be considered that, in engineering applications, the designer it not typically looking for the global optimum but for a local one, which is near to the original design. In order to find the latter, the more effective methodology of gradient based optimisation can be used, which is the one of interest in this thesis. The category of stochastic based optimisation escapes the scope of the present research then and therefore is not going to be discussed further. For some more detailed information, one could refer to [45, 27, 73, 33].

1.3 Gradient based optimisation

In an optimisation problem, the target is to find the (local or global) extremum of a cost function $\mathcal{F}$, the extremum being a minimum or a maximum. The logic of gradient–based optimisation is based on the following theorems[1] :

[1] The symbols $\searrow$ and $\nearrow$ represent continuously decreasing and continuously increasing function respectively

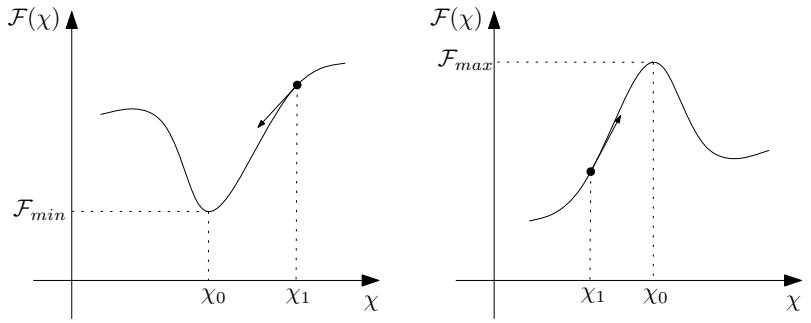


Figure 1.2: From the starting point χ_1 to the extremal χ_0 using gradients in 2D.

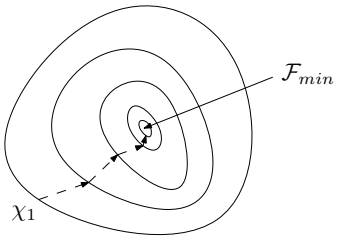


Figure 1.3: From the starting point χ_1 to the extremal $\mathcal{F}_{min}$ using gradients in 3D.

Theorem 1 *:*
If $\Phi : A \mapsto B$ *is a continuous function, then :*

- $\Phi \searrow \in (\chi_0, \chi_1) \Leftrightarrow \Phi^{'} \leq 0$
- $\Phi \nearrow \in (\chi_0, \chi_1) \Leftrightarrow \Phi^{'} \geq 0$

Theorem 2 *:*
If $\Phi : A \mapsto B$ *is a continuous function and* $[\Phi \searrow \in (\chi_1, \chi_0), \Phi \nearrow \in [\chi_0, \chi_2)]$ *or* $[\Phi \nearrow \in (\chi_1, \chi_0), \Phi \searrow \in [\chi_0, \chi_2)]$ *then* $\Phi^{'}(\chi_0) = 0$.

Based on these theorems, the gradient of the cost function $\mathcal{F}$ can be used to either "slide" down to the minimum or "climb" up to the maximum of $\mathcal{F}$, as shown in Fig. 1.2 and 1.3. Over the years, a number of methodologies to solve optimisation problems using gradients have been developed. These are going to be discussed in Chapter 4.Before exploring the gradient based optimisation methods further, it shall be mentioned that the latter are not always able to find the global minimum and might get trapped in a local one. This is an argument frequently used by researchers in the field of stochastic optimisation methods. For example, it would easier for a gradient based optimisation method to stop the optimisation process once a local minimum is reached in the "noisy" Schwefel function of Figure 1.5.

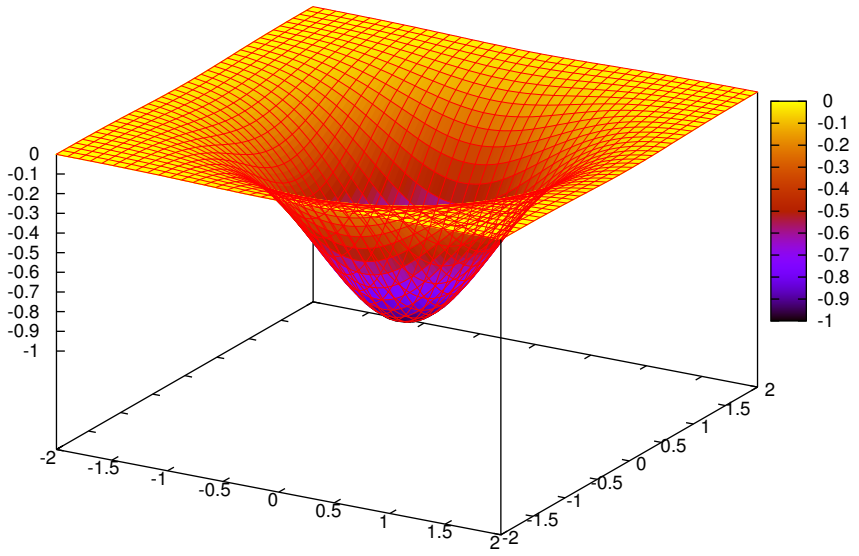


Figure 1.4: Example of a minimum in a three dimensional problem: minimum of the function $f(x, y) = -e^{-x^2} \cdot e^{-y^2}$.

Despite this problem though and the fact that there are ways to bypass this problem (such as multi point optimisation), it shall be kept in mind that in most engineering cases, local extrema are typically acceptable and not global ones. The target gradient based methods is to deliver the optimum that "leaves" close to a given solution. As an example, the design of a car shall be considered. The artistic designer is delivering a first shape of the car and then the engineer is asked to increase its aerodynamic performance, but without completely changing the shape. Therefore, the optimal shape for the given design will be somewhere close to the latter and a gradient based optimisation method can drive to it very fast. Especially in the case of adjoint based methods, this could mean even just a few evaluations. So, the restrictions of gradient based methods are known in this thesis, but the delivery of algorithms that can compute local optimal shapes fast is considered more important for engineering applications.

1.4 Optimisation in aerodynamics

There are several aspects in real world applications that can benefit from optimisation. Especially in the field of aerodynamics, engineering applications come in great variety. Some of them are airfoils and wings, airplane bodies, turbo machinery components (rotor/stator blades, channels, main body) helicopter blades, automotive external

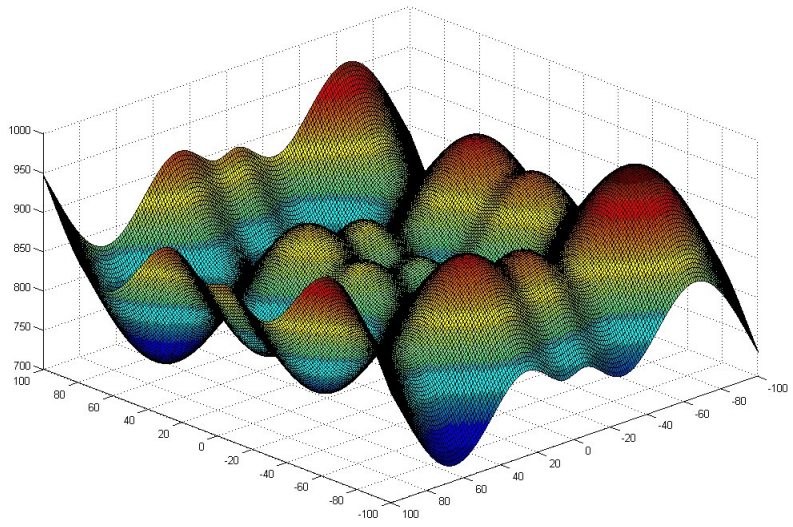


Figure 1.5: The Schwefel function.

aerodynamics, internal ducts, naval architecture and ship design, high speed trains, civil engineering structures, artificial implants, sports equipment etc. The list can be long extended and the applications become even more numerous, once the adjoint approach is considered. As it will be explained in Chapter 3, using the adjoint methodology, one can acquire the source of a behaviour and use that information to solve existing problems. For instance, an application of CFD and adjoint has been presented for bomb impact prediction by Professor Rainald Löhner, which makes obvious that the nature of applications, where the methodology can be applied, varies to a great extent. It is up to the scientists to encounter more and more applicable cases.

The biggest motivation to the derivation of a fast optimisation method is that it could remove the need of several years experience and "trial and error" experimentation. To make this point more clear, one can consider the developments in the aeronautical and automotive industry, illustrated in Figures 1.6 and 1.7. The evolution in the shape and the aerodynamic efficiency is more than obvious in the figures above. These changes though needed more than a century to come about and they were all based on highly costly experiments and a long chain of continuous error learning. All this progress, present in other fields of engineering and science as well of course, was "optimisation by hand", in the sense that there was no inverse design logic behind the development. The results therefore, always had to be an improvement to the currently existing applications and, even that improvement was due to even further and better change. One could say that, something better was always acquired but never the optimum. On the other hand, with

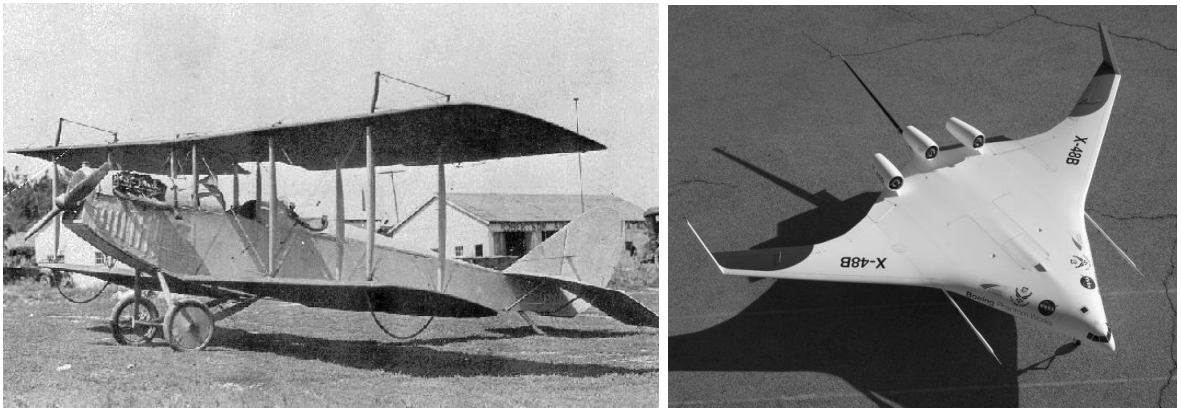


Figure 1.6: Aerodynamic evolution in airplanes. Left: The Robertson airplane (source: Digital Collections), Right: Boeing high efficiency concept airplane (source:Boeing)

Figure 1.7: Aerodynamic evolution in cars. Left: one of the first cars (source: Alaina Hoffmann), Right: Current concept "green" car (source: Digital Collections)

the adoption of optimisation techniques, the optimum of a desired application (at least in a local scale) can be found and within a few days or hours, with very high accuracy in the results. Such a possibility becomes of even more and more increasingly important in the last years, since the engineers are asked to deliver highly effective and nature friendly products. The use of optimisation in industry can help towards that approach and lead to airplanes, cars, etc. with very low fuel consumptions and a step towards a greener planet.

A lot of the effort in this research is placed not only in the development of such a methodology but also on the automated derivation and maintenance of the relative software. It is a common problem in CFD, that the effective maintenance of the source code can be difficult. This becomes even more complicated when, apart from the flow solver, there are sensitivity solvers as well, e.g. an adjoint solver. As it will be shown, with the use Algorithmic Differentiation and advanced scripting, the process of generating, validating and maintaining any sensitivity solver can be full automated and adapted to any change in the flow solver. This logic is applied not only to the adjoint solver but also in the direct differentiation one as well as two solvers of second derivative systems, the

direct differentiation over direct differentiation and the direct differentiation over adjoint.

In the flowing chapters, the logic behind gradient based optimisation using the methodologies above and Algorithmic Differentiation is going to be presented. The ways, in which such an approach can be implemented, are described in detail. Issues of performance acceleration are dealt with and the whole the whole theory is coded together to form the contribution to the in–house research code **mgOpt**. This software is applied to a number of cases, which are presented in the terms of flow, adjoint, optimisation and Hessian. In the end, the future work is outlined and the upcoming challenges and applications are discussed.

Chapter 2

The flow solver

2.1 General description

The basis of performing numerical aerodynamic shape optimisation is computing the flow quantities over a given geometry. This is performed by a stable, verified and validated flow solver, which can then be differentiated to provide sensitivities of the flow with respect to the design variables. Therefore, before discussing about derivatives and optimisation, the flow solver greater developed throughout this project is going to be discussed in this Chapter.

As a part of the software **mgOpt**, an in–house legacy computer program, the flow solver could, before the work of the author, compute first and second order accurate Euler (inviscid) flows for cases in two dimensions and using Roe's flux scheme, without limiters. It was previously developed by Dr Jens–Dominik Müller, Dr Paul Cusdin and Armen Jaworski [55], written in Fortran 77, who explored the first adjoint version of the software (by hand generation of adjoint via algorithmic differentiation) and presented results in [55, 56, 23, 25, 24]. The entire software has been further developed with the work of the author and the details are going to be presented.

In its current form, **mgOpt** is a two and three dimensional, vertex centred, first and second order accurate, hybrid element (triangles, quads, pyramids, prisms, tetrahedra, hexahedral) program, which makes use of Roe and AUSM^{+}_{up} flux routines. Various flux limiters are included (Barth, Venkatakrisnan and minmod) and the Spalart-Allmaras turbulence model has also been implemented.

Data is stored at the grid vertices and the flow residuals are accumulated in loops over edges. The solver, including the adjoint, tangent and others that will be discussed in later chapters, is computing the solutions in a iterative/re-computation manner, so there is no need for storage of the Jacobian matrix.

The source code is written in high level Fortran 90/95 (using modules, derived data

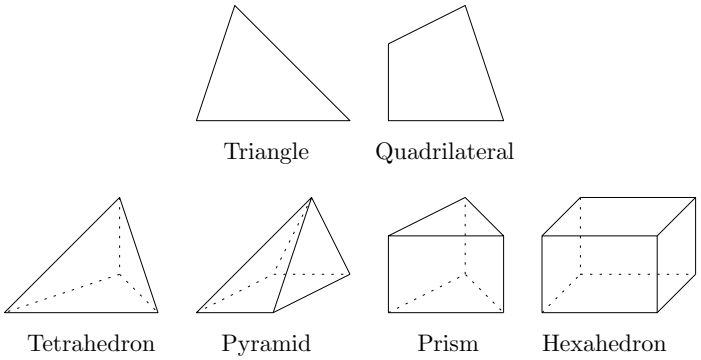


Figure 2.1: Supported discretisation elements.

types, pointers e.t.c.) and is linked with libraries in Fortran 90 (L-BFGS-B [111]), Fortran 77 (Tapenade [101]) and C (Tapenade [101]). For the compilation, GNU gfortran [40] and gcc [41] are used.

Having briefly discussed the basic characteristics of the software **mgOpt**, the next sections are devoted to the detailed description of the flow solver.

2.2 Spatial discretisation

The spatial discretisation logic used is edge-based and there is a variety of elements supported in **mgOpt**, characterising it as a *hybrid* solver. In its form before the authors research work, the software was supporting triangles and quadrilateral elements for geometries in two dimensions. This was extended in three dimensions and support was added for tetrahedra, pyramids, prisms and hexahedra. All the supported elements are illustrated in Figure 2.1.

Between the cell–centred (higher storage cost) and vertex–centred (higher compute cost) storage schemes, the latter is used because it allows the use of more accurate Dirichlet boundary conditions [38] and is convenient for mesh adaptivity (future implementation plan).

Once the vertex–centred logic has been employed, the control volumes can be formed by the **centroid** or the **median** dual, out of which the first is used [9]. An example is given in Figure 2.2. The centroid dual is chosen due to the following reasons :

1. It provides simpler control volumes (especially in three dimensional cases) compared to the **median**. A comparison of can already be made in the two dimensional example of Figure 2.3.

2. The median dual has higher storage requirements due to the higher number of flux faces.

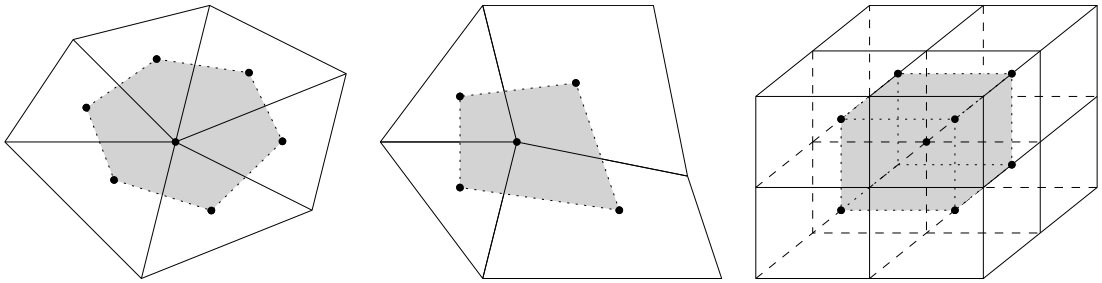


Figure 2.2: Examples of control volumes in two and three dimensions for the centroid dual scheme.

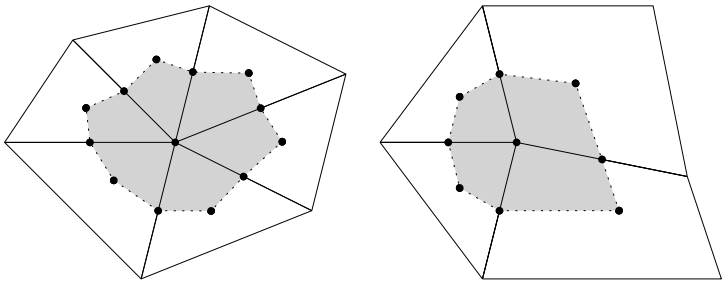


Figure 2.3: Examples of control volumes for the median dual scheme. The median dual control volume has two and four times the number of flux faces in 2D and 3D respectively, compared to the centroid dual (Figure 2.2). This can lead to very complicated and deformed control volumes.

Although the topic of the presented work is not mesh generation, credit should be given to the software used for that purpose, as they helped illustrate the research work. The open – source software `Gmsh` [36] was mainly used, as well as commercial products `Ansys Gambit` [5] and `Ansys ICEM-CFD` [6]. In two dimensional cases, the free – ware `ipol` [78] and `Delaundo` [76] were used for geometry preparation/manipulation.

2.3 Flux schemes

The flow solver developed provides solution to the compressible Navier–Stokes equations, which in a Cartesian coordinate system be written in differential form [12] as :

$$\frac{\partial \rho}{\partial t} + \frac{\partial (\rho u_x)}{\partial x} + \frac{\partial (\rho u_y)}{\partial y} + \frac{\partial (\rho u_z)}{\partial z} = 0 \tag{2.1}$$

$$\rho \left(\frac{\partial u_x}{\partial t} + u_x \frac{\partial u_x}{\partial x} + u_y \frac{\partial u_x}{\partial y} + u_z \frac{\partial u_x}{\partial z} \right) = -\frac{\partial p}{\partial x} - \frac{\partial \tau_{xx}}{\partial x} - \frac{\partial \tau_{yx}}{\partial y} - \frac{\partial \tau_{zx}}{\partial z} + F_x \tag{2.2}$$

$$\rho\left(\frac{\partial u_y}{\partial t}+u_x\frac{\partial u_y}{\partial x}+u_y\frac{\partial u_y}{\partial y}+u_z\frac{\partial u_y}{\partial z}\right) = -\frac{\partial p}{\partial y}-\frac{\partial \tau_{xy}}{\partial x}-\frac{\partial \tau_{yy}}{\partial y}-\frac{\partial \tau_{zy}}{\partial z}+F_y \quad (2.3)$$

$$\rho\left(\frac{\partial u_z}{\partial t}+u_x\frac{\partial u_z}{\partial x}+u_y\frac{\partial u_z}{\partial y}+u_z\frac{\partial u_z}{\partial z}\right) = -\frac{\partial p}{\partial z}-\frac{\partial \tau_{xz}}{\partial x}-\frac{\partial \tau_{yz}}{\partial y}-\frac{\partial \tau_{zz}}{\partial z}+F_z \quad (2.4)$$

$$\begin{aligned}\rho\left(\frac{\partial E_t}{\partial t}+\frac{\partial u_x E_t}{\partial x}+\frac{\partial u_y E_t}{\partial y}+\frac{\partial u_z E_t}{\partial z}\right) = &-\left[\frac{\partial (u_x p)}{\partial x}+\frac{\partial (u_y p)}{\partial y}+\frac{\partial (u_z p)}{\partial z}\right]\\ &-\frac{1}{Re\cdot Pr}\left(\frac{q_x}{x}+\frac{q_y}{y}+\frac{q_z}{z}\right)\\ &+\frac{1}{Re}\left[\frac{\partial}{\partial x}(u_x\tau_{xx}+u_y\tau_{xy}+u_z\tau_{xz})\right.\\ &+\frac{\partial}{\partial y}(u_x\tau_{xy}+u_y\tau_{yy}+u_z\tau_{yz})\\ &\left.+\frac{\partial}{\partial z}(u_x\tau_{xz}+u_y\tau_{yz}+u_z\tau_{zz})\right] \quad (2.5)\end{aligned}$$

where ρ is density, $\vec{U}=[u_x,u_y,u_z]$ velocity, t time, p pressure, $\vec{\tau}=[\tau_{ij}]\ \forall\ i,j=[x,y,z]$ the matrix of stress tensors and $\overrightarrow{F}=[F_x,F_y,F_z]$ the vector of body forces acting on the fluid. Re and Pr are the Reynolds (2.6) and Prandtl (2.7) numbers respectively and E_t the total energy (2.8). Lastly, $\overrightarrow{q}=[q_x,q_y,q_z]$ is the heat flux vector (2.9).

$$Re=\frac{\rho UL}{\mu}=\frac{UL}{\nu} \quad (2.6)$$

$$Pr=\frac{\nu}{\alpha}=\frac{c_p\mu}{k} \quad (2.7)$$

$$E_t=\rho e+\frac{1}{2}\rho(u_x^2+u_y^2+y_z^2) \quad (2.8)$$

$$q_i=-k\frac{\partial T}{\partial x_i}=-c_P\frac{\mu}{Pr}\frac{\partial T}{\partial x_i} \quad (2.9)$$

In the equations above, U is the mean velocity, μ the dynamic viscosity, ν the kinematic viscosity, α the thermal diffusivity, e the internal energy per unit mass for the fluid, c_P the specific heat and k the thermal conductivity.

In many of the results that are going to be presented in this and later chapters, the set of Euler equations are used, which are a simplification to the Navier–Stokes equations by assuming zero viscosity in the flow. Therefore, the equations to be solved take the form of (2.10).

The equations above can be solved by employing a finite volume scheme, which is going to be discussed in Sections 2.3.1 and 2.3.2. Two flux functions have been implemented for this reason: Roe's [95, 96] and the $AUSM^+_{up}$ [62, 63] scheme. Roe's scheme is a popular upwind flux-difference scheme, which is commonly used for compressible flows

and speeds of Mach number $Ma \geq 0.3$. The latter belongs to the greater family of the Advection Upstream Splitting Method (AUSM) and is able of solving efficiently even low speed compressible flows, where $Ma \geq 0.01$. In what follows, both schemes are going to be discussed.

$$\begin{cases} \frac{\partial \rho}{\partial t} + \frac{\partial(\rho u_x)}{\partial x} + \frac{\partial(\rho u_y)}{\partial y} + \frac{\partial(\rho u_z)}{\partial z} = 0 \\ \rho\left(\frac{\partial u_x}{\partial t} + u_x\frac{\partial u_x}{\partial x} + u_y\frac{\partial u_x}{\partial y} + u_z\frac{\partial u_x}{\partial z}\right) + \frac{\partial p}{\partial x} = 0 \\ \rho\left(\frac{\partial u_y}{\partial t} + u_x\frac{\partial u_y}{\partial x} + u_y\frac{\partial u_y}{\partial y} + u_z\frac{\partial u_y}{\partial z}\right) + \frac{\partial p}{\partial x} = 0 \\ \rho\left(\frac{\partial u_z}{\partial t} + u_x\frac{\partial u_z}{\partial x} + u_y\frac{\partial u_z}{\partial y} + u_z\frac{\partial u_z}{\partial z}\right) + \frac{\partial p}{\partial x} = 0 \\ \rho\left(\frac{\partial E_t}{\partial t} + \frac{\partial u_x E_t}{\partial x} + \frac{\partial u_y E_t}{\partial y} + \frac{\partial u_z E_t}{\partial z}\right) + \left[\frac{\partial(u_x p)}{\partial x} + \frac{\partial(u_y p)}{\partial y} + \frac{\partial(u_z p)}{\partial z}\right] = 0 \end{cases} \tag{2.10}$$

2.3.1 Roe's approximate Riemann solver

Roe's approximate Riemann solver [96, 95] belongs to the category of flux-difference splitting schemes, where the convective fluxes are evaluated on the faces of the control volumes using the left and right states. The idea was first introduced by Godunov [44] and the main logic is based on the Riemann problem (shock tube).

If $\vec{F}(\vec{U})$ is the flux vector, the flow equations can be written in the form of equation 2.11.

$$\begin{aligned} \frac{\partial \vec{U}}{\partial t} + \frac{\partial \vec{F}(\vec{U})}{\partial x} + \frac{\partial \vec{F}(\vec{U})}{\partial y} + \frac{\partial \vec{F}(\vec{U})}{\partial z} &= 0 \\ \Rightarrow \frac{\partial \vec{U}}{\partial t} + \frac{\partial \vec{F}(\vec{U})}{\partial \vec{U}}\frac{\partial \vec{U}}{\partial x} + \frac{\partial \vec{F}(\vec{U})}{\partial \vec{U}}\frac{\partial \vec{U}}{\partial y} + \frac{\partial \vec{F}(\vec{U})}{\partial \vec{U}}\frac{\partial \vec{U}}{\partial z} &= 0 \\ \Rightarrow \frac{\partial \vec{U}}{\partial t} + \mathrm{A}(\vec{U})\frac{\partial \vec{U}}{\partial x} + \mathrm{A}(\vec{U})\frac{\partial \vec{U}}{\partial y} + \mathrm{A}(\vec{U})\frac{\partial \vec{U}}{\partial z} &= 0 \end{aligned} \tag{2.11}$$

In [96], Roe was in search of constructing a matrix $\mathbf{A}_{Roe}$, which would satisfy equation (2.11) and will have the following properties :

1. It achieves a linear mapping from $\vec{U}$ to $\vec{F}$.
2. When $\vec{U}_L \to \vec{U}_R \to \vec{U}$ then $\mathbf{A}_{Roe}(\vec{U}_L, \vec{U}_R) \to \mathbf{A}(\vec{U})$, where $\mathbf{A} = \frac{\partial \vec{F}}{\partial \vec{U}}$ the flux Jacobian.
3. $\forall \vec{U}_L, \vec{U}_R$: $\mathbf{A}_{Roe}(\vec{U}_l, \vec{U}_R) \times (\vec{U}_l - \vec{U}_R) = \vec{F}_l, \vec{F}_R$
4. The eigenvectors of $\mathbf{A}_{Roe}$ are linearly independent.

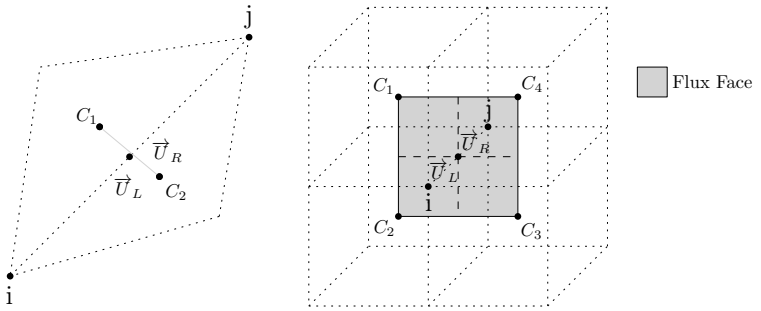


Figure 2.4: Flux face in a centroid centred scheme between the nodes i and j. L and R denote left and right state respectively and C_k centroids.

The flux through a flux face (Figure 2.4, centroid centred dual volume), is given as :

$$\vec{F}_{1/2}(\vec{U}_L, \vec{U}_R) = \mathbf{A}_{Roe,1/2}(\vec{U}_L, \vec{U}_R)\vec{\widehat{U}} \tag{2.12}$$

where $\vec{\widehat{U}}$ the so–called Roe averaged variables. The subscripts $1/2, L$ and R denote the flux face, left and right state respectively.

Knowing the convective flux Jacobian $\mathbf{A}_c$ (matrix that contains the derivatives of the convective fluxes with respect to the conservative variables $\partial\vec{F}^c/\partial\vec{U}$), Roe's Jacobian matrix $\mathbf{A}_{Roe}$ can be similarly built by using the *Roe averaged variables* instead :

$$\widehat{\rho} = \sqrt{\rho_L \rho_R} \tag{2.13}$$

$$\widehat{u} = \frac{u_L\sqrt{\rho}_L + u_R\sqrt{\rho}_R}{\sqrt{\rho}_L + \sqrt{\rho}_R} \tag{2.14}$$

$$\widehat{v} = \frac{v_L\sqrt{\rho}_L + v_R\sqrt{\rho}_R}{\sqrt{\rho}_L + \sqrt{\rho}_R} \tag{2.15}$$

$$\widehat{w} = \frac{w_L\sqrt{\rho}_L + w_R\sqrt{\rho}_R}{\sqrt{\rho}_L + \sqrt{\rho}_R} \tag{2.16}$$

$$\widehat{H} = \frac{H_L\sqrt{\rho}_L + H_R\sqrt{\rho}_R}{\sqrt{\rho}_L + \sqrt{\rho}_R} \tag{2.17}$$

$$\widehat{c} = \sqrt{(\gamma-1)(\widehat{H} - \widehat{q}^2/2)} \tag{2.18}$$

$$\widehat{V} = \widehat{u}n_x + \widehat{v}n_y + \widehat{w}n_z \tag{2.19}$$

$$\widehat{q}^2 = \widehat{u}^2 + \widehat{v}^2 + \widehat{w}^2 \tag{2.20}$$

For further information on Roe's scheme and a more in depth analysis, one can refer to [75].

2.3.2 AUSM$^+_{up}$ scheme

The $AUSM^+_{up}$ flux scheme belongs to the greater family of the Advection Upstream Splitting Method (AUSM), which was first introduced by Liou [64, 61, 62]. AUSM$^+_{up}$ [63] is the most recent and most effective of all other alternations of the AUSM family, as it can handle low speed compressible flows quite accurately. It was implemented in **mgOpt** with the help of the PhD student Shenren Xu. This flux function was adopted in this research for the following reasons :

- The time–stepping algorithm of compressible discretisations is fully coupled, i.e. all equations are dealt with in the same way. This makes it simpler to devise efficient time-stepping schemes.
- It was within the research plans to compute second derivatives using adjoint methods 3, so the methodology was first developed using the simpler time-stepping of compressible flows. In the future Algorithm Differentiation for second derivatives of incompressible SIMPLE schemes [88] could be developed.
- The $AUSM^+_{up}$ flux helps to lower the threshold of the Mach number for which compressible discretisations are applicable.

In the $AUSM^+_{up}$ scheme, the flux $\vec{F}_{1/2}$ though a flux face (see Figure 2.4), consists of two parts, the convective and the pressure flux, as expressed in (2.21). The subscript 1/2 denotes the flux interface.

$$\vec{F} = \dot{m}\vec{\phi} + \vec{P} \tag{2.21}$$

The mass and pressure fluxes, $\dot{m}$ and $\vec{P}$ are given by (2.22) and (2.23).

$$\dot{m} = \rho M_{1/2} a_{1/2} \tag{2.22}$$

$$\vec{P} = \begin{bmatrix} 0 \\ p_{1/2}n_x \\ p_{1/2}n_y \\ p_{1/2}n_z \\ 0 \end{bmatrix} \tag{2.23}$$

$\alpha_{1/2}$ and $p_{1/2}$ are the speed of sound and pressure at the interface and $\vec{n} = [n_x, n_y, n_z]$ is the unit normal of the flux face.

The vector $\vec{\phi}$ is given as $\vec{\phi} = [1, u, v, w, H]$. In this formulation, the density ρ and the vector $\vec{\phi}$ directly adopt the values of either the left or the right states depending on

the sign of $M_{1/2}$. For this reason, the main focus in $AUSM^+_{up}$ is the computation of the Mach number $M_{1/2}$, the speed of sound $a_{1/2}$ and the pressure $p_{1/2}$ at the interface.

The speed of sound is defined by (2.24) :

$$a_{1/2} = \min(\hat{a}_L, \hat{a}_R) \tag{2.24}$$

where a_L and a_R the speed of sound left and right of the interface are given as :

$$\begin{cases} \hat{a}_L = a^{*^2}/\max(a^*, V_L) \\ \hat{a}_R = a^{*^2}/\max(a^*, -V_R) \end{cases} \tag{2.25}$$

The *critical* speed of sound (when the local Mach number is equal to unity) at the interface a^{*^2} can be computed using the definition of total enthalpy.

$$\begin{aligned} H_t = \frac{a^2}{\gamma - 1} + \frac{1}{2}V^2 = \frac{(\gamma+1)a^{*^2}}{2(\gamma-1)} \\ \Rightarrow a^* = \sqrt{\frac{2(\gamma-1)H_t}{\gamma-1}} = \sqrt{\frac{2a^2 + (\gamma-1)V^2}{\gamma+1}} \end{aligned} \tag{2.26}$$

In (2.25), V_R and V_L are the contra-variant velocities $V_{L,R} = [\vec{u} \cdot \vec{n}]_{L,R}$ on the right and left states respectively.

The Mach number $M_{1/2}$ at the interface is defined by equation (2.27) :

$$M_{1/2} = \mathcal{M}^+_{(4)}(M_L) + \mathcal{M}^-_{(4)}(M_R) - \frac{K_p}{f_a(M_o)} \cdot \max(1 - \sigma\bar{M}^2, 0) \cdot \frac{p_R - p_L}{\rho_{1/2} a^2_{1/2}} \tag{2.27}$$

where :

$$\begin{cases} M_L = \dfrac{u_L}{a_{1/2}} \ , \ M_R = \dfrac{u_R}{a_{1/2}} \ , \ \bar{M}^2 = \dfrac{u_L^2 + u_R^2}{2a^2_{1/2}} \\ \\ M_o^2 = \min(1, \max(\bar{M}^2, M_\infty^2)) \ , \ f_a(M_o) = M_o(2 - M_o) \\ \\ \rho_{1/2} = (\rho_L + \rho_R)/2 \end{cases} \tag{2.28}$$

$$\begin{cases} \mathcal{M}^{\pm}_{(4)}(M) = \begin{cases} \mathcal{M}^{\pm}_{(1)} & \text{if } |M| \geq 1 \\ \mathcal{M}^{\pm}_{(2)}(1 \mp 16\beta\mathcal{M}^{\mp}_{(2)}) & \text{otherwise.} \end{cases} \\ \\ \mathcal{M}^{\pm}_{(1)}(M) = (M \pm |M|)/2 \ , \ \mathcal{M}^{\pm}_{(2)}(M) = \pm(M \pm 1)^2/4 \end{cases} \tag{2.29}$$

The mass flux at the interface $\dot{m}_{1/2}$ can then be defined as :

$$\dot{m}_{1/2} = \begin{cases} \rho_L \cdot M_{1/2} \cdot a_{1/2} & \text{if } M_{1/2} > 0 \\ \rho_R \cdot M_{1/2} \cdot a_{1/2} & \text{otherwise.} \end{cases} \tag{2.30}$$

The pressure flux at the interface $p_{1/2}$ is defined in (2.31).

$$p_{1/2} = \mathcal{P}^{+}_{(5)}(M_L)p_L + \mathcal{P}^{-}_{(5)}(M_R)p_R + p_u \tag{2.31}$$

where p_u and $M^{pm}_{(M)}$ are given by (2.32) and (2.33) respectively.

$$p_u = -K_u\mathcal{P}^{+}_{(5)}(M_L)\mathcal{P}^{-}_{(5)}(M_R)(\rho_L + \rho_R)(f_a a_{1/2})(u_R - u_L) \tag{2.32}$$

$$\mathcal{P}^{\pm}_{(5)}(M) = \begin{cases} \mathcal{M}^{\pm}_{(1)}/M & \text{if } |M| \geq 1, \\ \mathcal{M}^{\pm}_{(2)}[(\pm 2 - M) \mp 16\alpha M\mathcal{M}^{\mp}_{(2)}] & \text{otherwise.} \end{cases} \tag{2.33}$$

Finally, the flux at interface can be defined by (2.34) :

$$\vec{F}_{1/2} = \dot{m}_{1/2} \begin{cases} \vec{\phi}_L & \text{if } \dot{m}_{1/2} > 0 \\ \vec{\phi}_R & \text{otherwise} \end{cases} + \vec{P}_{1/2} \tag{2.34}$$

where :

$$\vec{\phi} = [1, u, v, w, H]^T \tag{2.35}$$

The values of the parameters α. β, σ, K_p and K_p above are taken from [63].

$$\begin{cases} \alpha = \dfrac{3}{16}(-4 + 5f_a^2) \\ \beta = 0.125 \\ \sigma = 1.0 \\ K_p = 0.25 \\ K_u = 0.75 \end{cases} \tag{2.36}$$

Once a flux function, such as Roe's (Section 2.3.1) or $AUSM^+_{up}$ is available, issues of flux discretisation accuracy can be examined. This is associated with the assumptions of variations of the flow variables within each control volume and will be discussed in Section 2.4.

2.4 Solution accuracy

The accuracy of a numerical flow solution in space is related with the flow variation assumptions within each control volumes. In the presented research first and second order accurate computations where implemented, whereas higher order methods escape its scope.

In first order accurate calculations, the state variables are piecewise–constant over the control volumes and only first–order neighbours (connected via an edge of the grid in the case of vertex–centred dual scheme) are considered in the computational stencil. Referring to Figure 2.4, $\vec{U}_{L,R}$ are defined as :

$$\vec{U}_L = \vec{U}_i \tag{2.37}$$

$$\vec{U}_R = \vec{U}_j \tag{2.38}$$

For second order accurate computations, linear change over the control volumes is assumed. The computational stencil includes the original computational nodes plus the extrapolated values at the flux interface and the solution becomes second order accurate in space. This implies that, the solution has to be reconstructed in order to calculate the left and right flow states of a control volume face, which are needed in upwind schemes. The most commonly used reconstruction techniques are :

1. Reconstruction based on the MUSCL approach [102].
2. Piecewise linear reconstruction [7].
3. Nodal weighting based linear reconstruction [34].

Based on the data structure of **mgOpt** and wanting to keep the memory and runtime cost as low as possible, the method chosen was the piecewise linear reconstruction. For this and following the method of Barth and Jespersen [7], the left and right state for a centroid-dual scheme can be calculated respectively from (2.39) and (2.39) :

$$\begin{aligned} U_L &= U_i + \Psi_i \cdot \nabla U_i \cdot \vec{r}_L \\ U_R &= U_j + \Psi_j \cdot \nabla U_j \cdot \vec{r}_R \end{aligned} \tag{2.39}$$

where $U_{i,j}$ are the flow variables, $\nabla U_{i,j} = \left[\frac{\partial U}{\partial x}, \frac{\partial U}{\partial y}, \frac{\partial U}{\partial z}\right]^T_{i,j}$ the spatial gradients of the flow variables and $\Psi_{i,j}$ the values of the limiter function at nodes i and j respectively. The vectors $\vec{r}_{i,j}$ connect the grid nodes i and j with the control volume face midpoint, Figure 2.5. The gradients and limiters involved in Equation 2.39 are discussed in the

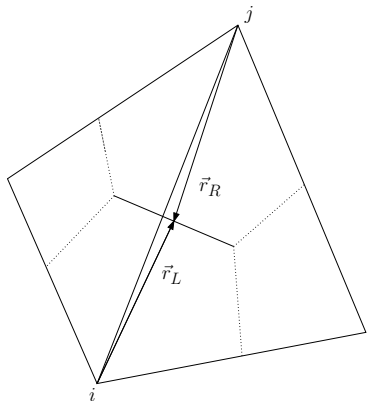


Figure 2.5: Vectors from the vertices to the flux face midpoint.

next subsections.

2.4.1 Gradient computation

The two most common methodologies to calculate the gradients are those using the Green - Gauss theorem [7] or the least squares theory [9, 8]. For this research, the first was implemented by the author, as it is computationally less expensive and can also be used for the calculation of the size of control volumes.

The area around the vertex, where the gradient must be computed (vertex–centred logic), has the form of Figure 2.6.

The gradient within this region can be calculated using the the Green-Gauss formula over the area Ω_0, Equation (2.40) :

$$\int_{\Omega_0} \nabla u d\alpha = \oint_{\partial\omega_0} u\vec{n} dl \tag{2.40}$$

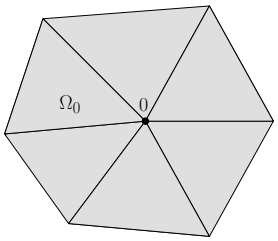


Figure 2.6: The area around the vertex of interest.

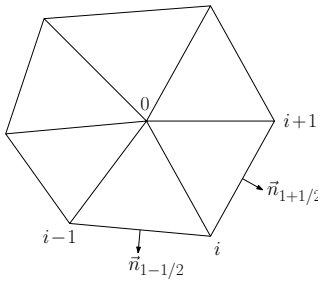


Figure 2.7: The application of the Green-Gauss theorem.

Following the approach of Barth [9] and the notation of Figure 2.7, the right hand side of (2.40) can be approximated by trapezoidal quadrature, which is exact for linear variations, Equation 2.41.

$$\oint_{\partial\Omega_0} u\vec{n}dl \approx \oint_{\partial\Omega_0} u^h\vec{n}dl = \sum_{i=1}^{K} \frac{1}{2}\left(u_i^h + u_{i+1}^h\right)\cdot\vec{n}_{i+1/2} \tag{2.41}$$

In (2.41), h stands for the linearly varying values of the state variable at the vertices and K is the total number of the vertices neighboring with node 0. The vector $\vec{n}_{i+1}$ is perpendicular to the edge $(i, i+1)$, with magnitude equal to the length of this edge. With rearrangement of the terms on the right hand side, (2.41) can be written in the form of (2.42).

$$\oint_{\partial\Omega_0} u^h\vec{n}dl = \sum_{i=1}^{K} \frac{1}{2}u_i^h\left(\vec{n}_{i-1/2} + \vec{n}_{i+1/2}\right) \tag{2.42}$$

Since the gradient of a constant function is zero, any constant solution can be added to (2.42) without changing the result. Therefore, one can add the value of u_0^h, which leads to Equation (2.43).

$$\oint_{\partial\Omega_0} u^h\vec{n}dl = \sum_{i=1}^{K} \frac{1}{2}\left(u_0^h + u_i^h\right)\left(\vec{n}_{i-1/2} + \vec{n}_{i+1/2}\right) \tag{2.43}$$

For any closed path though, $\oint \vec{n}dl = \oint d\vec{n} = 0$, which implies the relation of Equa-

tion (2.44), for any path connecting the vertices $(i-1)$ and $(i+1)$.

$$\vec{n}_{1-1/2} + \vec{n}_{1+1/2} = \int_{i-1}^{i+1} d\vec{n} \tag{2.44}$$

This path integral represents a vector parallel to and three times the magnitude of any vector $\vec{n}$ obtained by computing the integral for any simple path connecting the centroids of the two triangles, which share the edge $(0,i)$: $\int_{i-1}^{i+1} d\vec{n} = 3\int_{i'-1}^{i} d\vec{n} = 3\vec{n}_{0i}$. Using this, (2.43) takes the form of (2.45).

$$\oint_{\partial\Omega_0} u^h \vec{n} dl = \sum_{i=1}^{K} \frac{3}{2}\left(u_0^h + u_i^h\right)\vec{n}_{0i} \tag{2.45}$$

Using (2.45), the vertex lumped average gradient at vertex can be computed by (2.46).

$$\left(\nabla u^h\right)_0 = \frac{3}{A_{\Omega_0}} \sum_{i=1}^{K} \frac{1}{2}\left(u_0^h + u_i^h\right)\vec{n}_{0i} \tag{2.46}$$

For a function with known gradients (i.e. a linear function), the control volumes can be computed via (2.46), Equation (2.47).

$$.A_{\Omega_0} = \frac{3}{\left(\nabla u^h\right)_0} \sum_{i=1}^{K} \frac{1}{2}\left(u_0^h + u_i^h\right)\vec{n}_{0i} \tag{2.47}$$

2.4.2 Flux limiters

In second or higher order spatial discretisation, oscillatory solutions can occur in regions with high gradient, e.g. around shock waves. This problem is dealt with by using *limiter* functions, which preserve the monotonicity of a given function. These functions take values between zero and one and limit the reconstruction in areas that discontinuities may appear. In strong discontinuities they receive the value of zero, giving a first order upwind scheme that guarantees monotonicity (see Equation (2.39)), and the value of one at smooth flow regions. For three dimensional unstructured grids, the most established limiters are Barth and Jespersen's limiter [7], Venkatakrishnan's limiter [105, 104] and the minmod limiter [95]. In **mgOpt**, the first two were implemented.

2.4.2.1 Barth and Jespersen's limiter

Barth and Jespersen's limiter [7] was one of the first to be conceived and proposed for unstructured grids. It presents low complexity in implementation and this is the reason for its popularity in the Computational Fluid Dynamics community. For the centroid dual scheme that **mgOpt** is using, it is given by equation (2.48).

$$\Psi_i = min_j \begin{cases} min\left(1, \frac{U_{max}-U_i}{\Delta_2}\right) & \text{if } \Delta_2 > 0 \\ min\left(1, \frac{U_{min}-U_i}{\Delta_2}\right) & \text{if } \Delta_2 < 0 \\ 1 & \text{if } \Delta_2 = 0 \end{cases} \tag{2.48}$$

where :

$$\begin{aligned} \Delta_2 &= \frac{\nabla U_i \cdot A \cdot \overrightarrow{n}_{ff}}{2} \\ U_{max} &= max(U_i, max(U_{j,\forall j})) \\ U_{min} &= min(U_i, min(U_{j,\forall j})) \end{aligned} \tag{2.49}$$

The vector $\overrightarrow{n}_{ff}$ is the normal vector of the flux face between the nodes i and j (Figure 2.5) and A is the area of the face.

Despite the simplicity of this limiter, it has been observed that it can be activated in smooth flow regions, when high levels of numerical noise appear. This can lead to solution inaccuracy and prevent from reaching lower convergence levels [1, 104]. Also, the limiter has proven to produce high levels of dissipation. For this reasons, the more efficient limiter of Venkatakrisnan was also implemented and is presented in the next paragraph.

2.4.2.2 Venkatakrishnan's limiter

Venkatakrishnan's limiter [104, 105] is designed to overcome the problems of Barth's limiter. It is given as :

$$\Psi_i = min_j \begin{cases} \frac{1}{\Delta_2} \frac{\left(\Delta_{1,max}^2+\epsilon^2\right)\Delta_2+2\Delta_2^2\Delta_{1,max}}{\Delta_{1,max}^2+2\Delta_2^2+\Delta_{1,max}\Delta_2+\epsilon^2} & \text{if } \Delta_2 > 0 \\ \frac{1}{\Delta_2} \frac{\left(\Delta_{1,min}^2+\epsilon^2\right)\Delta_2+2\Delta_2^2\Delta_{1,min}}{\Delta_{1,min}^2+2\Delta_2^2+\Delta_{1,min}\Delta_2+\epsilon^2} & \text{if } \Delta_2 < 0 \\ 1 & \text{if } \Delta_2 = 0 \end{cases} \tag{2.50}$$

The variations $\Delta_{1,min}$ and $\Delta_{1,max}$ are given by (2.51).

$$\begin{aligned} \Delta_{1,min} &= U_{min} - U_i \\ \Delta_{1,max} &= U_{max} - U_i \end{aligned} \tag{2.51}$$

where $\vec{U}_i$ the vector of state variables at the vertex i, at which the limiter is being calculated, and U_{min} and U_{max} the minimum and maximum values respectively of all surrounding nodes j, including node i, as discribed in (2.52).

$$\begin{aligned} U_{min} &= min\left(U_i, min_j U_j\right) \\ U_{max} &= max\left(U_i, max_j U_j\right) \end{aligned} \tag{2.52}$$

For the centroid dual (Section 2.2), the variable Δ_2 is defined as :

$$\Delta_2 = \frac{1}{2}\left(\nabla U_i \cdot \vec{r}_{face_{ij}}\right) \tag{2.53}$$

where $\vec{r}_{face_{ij}}$ are the vector from node i to the midpoint of the flux face between the nodes i and j (as shown in figure 2.5).

Finally, the parameter ϵ^2 controls the aggressiveness of the limiter. For ϵ^2 equal to zero, the flow field becomes piecewise–constant everywhere, with the risk of stalling the convergence of the solution. For large values limiters approach unity, neutralising the limiter function. In practice, it was found [105], that this variable should have a value proportional to a local length scale :

$$\epsilon^2 = (k \cdot \Delta h)^3 \tag{2.54}$$

where k is a constant and Δh could be, for example, the square root of the area in 2D or the cube root of the volume in 3D. Practically, for values of k over 50, the limiter becomes neutralised.

Apart from the higher accuracy that Venkatakrishnan's limiter provides, compared to Barth's, its key design feature is the smooth switching, i.e. it reduces continuous switching on/off of the limiter and hence gives better convergence. This should lead to better differentiability, which will be performed in following chapters.

2.5 Temporal discretisation

In sections 2.3.1 and 2.3.2, the two different flux functions and the solution accuracy improvement methods used this study have been discussed. Once these are in place, a time stepping scheme needs to be employed so that the flow equations are solved (see time dependent terms in the flow governing equations of Section 2.3). In the case of steady flows, which will be the subject of the present research, these are actually pseudo time–stepping schemes, which drive the residuals to zero. These methods divide into two different subcategories, the implicit and the explicit time–stepping schemes. Only the

latter will be considered in this study, due to its implementation simplicity. For more information on the implicit time stepping schemes, the reader is referred to [12], Chapter 6.

Out of the explicit time-stepping, the most commonly used methodologies are those of multi-stage Runge-Kutta (R-K) and hybrid multi-stage scheme, which is an alteration of the first one. The hybrid multi-stage scheme [66, 71] is mostly applied to central discretisation schemes and is not considered this study.

Jameson's multi-stage Runge-Kutta (R-K) scheme [53] is commonly used in upwind schemes (such as the ones in this research) and its main logic is to divide the solution update in a number of steps.

Both Euler and Navier–Stokes equations (Section 2.3) can be summarised as :

$$\frac{d(\Omega \cdot \bar{M} \cdot \vec{Q})_i}{dt} + \vec{R}_i \tag{2.55}$$

where Ω the volume of the control volume i, $\bar{M}$ the mass matrix, $\vec{Q}$ and $\vec{R}_i$ the vectors of the conservative variables and residuals respectively. Considering a Runge–Kutta multi-stage scheme of n steps, the solution would be updated in consequent steps, as described by (2.56).

$$\begin{aligned}
\vec{Q}_i^1 &= \vec{Q}_i^0 - \lambda_1 \frac{\Delta t_i}{\Omega_i} \vec{R}_i^0 \\
\vec{Q}_i^2 &= \vec{Q}_i^0 - \lambda_2 \frac{\Delta t_i}{\Omega_i} \vec{R}_i^1 \\
&\dots \\
\vec{Q}_i^n &= \vec{Q}_i^0 - \lambda_n \frac{\Delta t_i}{\Omega_i} \vec{R}_i^n
\end{aligned} \tag{2.56}$$

where λ_i are the stage coefficients. The residuals R_i^0 and the solution $\vec{Q}_i^0$ are either the ones from the previous R-K or the initial ones. Different values have been proposed in the literature for the stage coefficients λ_i. This study uses the optimised coefficients given in [103], which are repeated in Table 2.1. It shall be noted that, λ_n must hold the value of one. Also, the scheme is second order accurate in time only if $\lambda_{n-1} \simeq 0.5$. For first order in-time accuracy the equivalent coefficients of Table 2.1 must be used in the case of strong discontinuities in the flow domain, such as shocks, for the same reasons that the limiter functions are used (see Section 2.4.2), that is reduce the accuracy of the scheme to first order. In this way, despite the spatial discretisation, oscilations or even stalling in the convergence of the solution are avoided.

The time step Δt_i for every control volume i must satisfy the *Courant-Friedricks-Lewy* (CFL) condition [21] for the scheme to remain stable. According to the CFL condition,

	first order			second order		
stages	3	4	5	3	4	5
CFL	1.5	2.0	2.5	0.69	0.92	1.15
λ_1	0.1481	0.0833	0.0533	0.1918	0.1084	0.0695
λ_2	0.4000	0.2069	0.1263	0.4929	0.2602	0.1602
λ_3	1.0000	0.4265	0.2375	1.0000	0.5052	0.2898
λ_4		1.0000	0.4414		1.0000	0.5060
λ_5			1.0000			1.0000

Table 2.1: Runge-Kutta optimisated coefficients for first and second order spatial discretisation schemes [103].

the domain of dependence of the partial differential equation must be included in the dependent domain of the numerical method for the numerical method to be stable. In the case of explicit time–stepping schemes, the latter means that the time–step for each control volume needs to be smaller or at maximum equal to the time required for the transport of information across the stencil of the spatial discretisation.

This is only a necessary condition. It is only also a sufficient condition for forward-Euler time-stepping. The stability limit for R-K multi-stage will be lower. Moreover, this only applies to the convective limit.

Several methods have been proposed for the satisfaction of this condition, but the one used in this study is the one proposed in [71]. The maximum time–step for each control volume i is computed by (2.57).

$$\Delta t_i = CFL \cdot \frac{\Omega_i}{(\Psi_{con} + \beta \Psi_{vis})_i} \tag{2.57}$$

In (2.57), Ψ_{con} and Ψ_{vis} represent the convective and viscous spectral radii and are computed over all flux faces of the control volumes, by (2.58) and (2.59).

$$\Psi^i_{con} = \sum_{j=1}^{N_F} (|\vec{u} \cdot \vec{n}|_{ij} + c_{ij}) \Delta S_{ij} \tag{2.58}$$

$$\Psi^i_{vis} = \frac{1}{\Omega_i} \sum_{j=1}^{N_F} \left[\max\left(\frac{4}{3\rho_{ij}}, \frac{\gamma_{ij}}{\rho_{ij}} \right) \left(\frac{\mu_{Lam}}{Pr_{Lam}} + \frac{\mu_{Tur}}{Pr_{Tur}} \right) \Delta S^2_{ij} \right] \tag{2.59}$$

Wherever the subscript ij is used, arithmetic averages of the equivalent variable are implied. N_F are the number of flux faces of the control volume i, ΔS_{ij} the area of the flux face j of the control volume i and c_{ij} the speed of sound at the flux face, takes as an average between the neighbouring control volumes at the flux face ij. The rest of the variables are defined as earlier in this chapter, with the subscripts ij, Lam (laminar) and

Tur (turbulent) used, when appropriate. The variable β takes values between one and four ($1 \leq \beta \leq 4$) for viscous cases and the value of zero for inviscid cases [12].

The next section describes boundary conditions, which conclude the mathematical problem of the partial differential equations used in this study. Out of the various boundary conditions in literature, a few have been selected and implemented, according to the needs of the flow problems examined.

2.6 Boundary Conditions

A set of partial differential equations can only be solved when well–posed boundary conditions are applied. These impose specific flow variables or flow variable properties on the boundaries of the computational domain and vary in types.

In **mgOpt**, the logic of the "ghost" cell methodology is used. This is associated with the following :

- Association of a fictitious "ghost" state with each boundary vertex.
- Appropriate modification of the state from the state of the boundary node, depending on the type of boundary condition.
- Same flux computation invoking as on the internal flux interfaces.
- The above lead to correct weak integral of the residual, but with the boundary vertex representing the integral over the control volume, not the boundary value.

An example of this logic is presented in Figure 2.8. In the next paragraphs, the boundary conditions implemented in **mgOpt** for the purposes of the present study are going to be discussed.

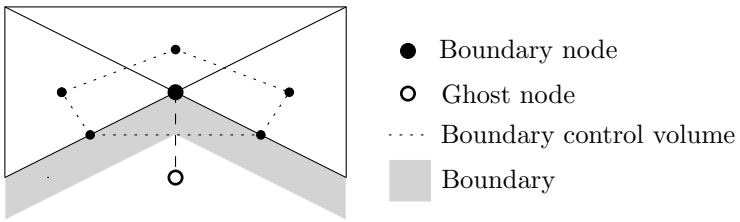


Figure 2.8: Example of a ghost node near the boundary. This node will be assigned with the "right" flow variables so that it consists the right or left state at the boundary flux face (Figure 2.4).

2.6.1 Subsonic far–field

The *subsonic far–field* boundary condition for upwinded flux functions imposes a specified state on boundaries very far from a solid geometry, e.g. far from the body of an airplane. On such boundaries, the flow is supposed to have returned to free–stream conditions, after interacting with a surface. If $\overrightarrow{Q}_{pr}$ and $\overrightarrow{Q}_{con}$ the vectors of primitive and conservative variables respectively, then this boundary condition can be expressed by (2.60) or (2.61).

$$\overrightarrow{Q}^{pr}_{Ghost} = \overrightarrow{Q}^{pr}_{farfield} \qquad \text{or} \qquad \overrightarrow{Q}^{con}_{Ghost} = \overrightarrow{Q}^{con}_{farfield} \tag{2.60}$$

$$\Rightarrow \begin{bmatrix} \rho \\ u \\ v \\ w \\ P \end{bmatrix} = \begin{bmatrix} \rho \\ u \\ v \\ w \\ P \end{bmatrix}_{farfield} \qquad \text{or} \qquad \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho E \end{bmatrix} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho E \end{bmatrix}_{farfield} \tag{2.61}$$

In equations (2.60) and (2.61) ρ represents density, $[u, v, w]$ the $[x, y, z]$ components of velocity.

2.6.2 Slip wall

The *slip wall* boundary condition is used in inviscid flow cases for solving the Euler equations and neglects any friction phenomena. Referring to Figure 2.8 and for such type of boundaries, the condition takes the form of (2.63), for primitive or conservative variables respectively.

$$\begin{bmatrix} \rho \\ u \\ v \\ w \\ P \end{bmatrix}_{Ghost} = \begin{bmatrix} \rho \\ u^t_x - u^n_x \\ v^t_x - v^n_x \\ w^t_x - w^n_x \\ P \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho E \end{bmatrix}_{Ghost} = \begin{bmatrix} \rho \\ \rho(u^t_x - u^n_x) \\ \rho(v^t_x - v^n_x) \\ \rho(w^t_x - w^n_x) \\ \rho E \end{bmatrix} \tag{2.62}$$

where $[u, v, w]^t_i$ and $[u, v, w]^n_i$ the tangential and normal to the wall components of the velocity in $i = [x, y, z]$ directions.

2.6.3 No slip wall

For viscous flow cases, the wall forces have to be taken into consideration. Therefore, for laminar of turbulent cases over solid walls a *no slip* boundary condition is needed. This

can be applied in the form of Equation (2.63) implies.

$$\begin{bmatrix} \rho \\ u \\ v \\ w \\ P \end{bmatrix}_{Ghost} = \begin{bmatrix} \rho \\ u \\ v \\ w \\ P \end{bmatrix}_{BN} \equiv \begin{bmatrix} \rho_{BN} \\ 0 \\ 0 \\ 0 \\ P_{BN} \end{bmatrix} \tag{2.63}$$

where BN stands for boundary node.

2.6.4 Subsonic inlet

The boundary condition of *subsonic inlet* [108] imposes four (in 3D) characteristics (density ρ and velocity $\vec{U}$), while it receives pressure from the interior domain. Mathematically, the boundary condition is expressed by equation (2.64).

$$\begin{bmatrix} \rho \\ u \\ v \\ w \\ P \end{bmatrix}_{Ghost} = \begin{bmatrix} \rho \\ u \\ v \\ w \\ 0 \end{bmatrix}_{BN} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \frac{P_0}{\gamma-1} + \frac{u^2+v^2+w^2}{2\rho} \end{bmatrix} \tag{2.64}$$

In equation (2.64), γ is the adiabatic ratio of specific heats and P_0 the pressure at the outlet. The difference of this boundary condition, compared to the subsonic far–field, is the computation of the pressure from the interior domain, which removes the need for the boundary to be far from solid surfaces. Because of this, the boundary condition is typically used ducted flows, such as intake pipes and turbo–machinery cascades.

2.6.5 Subsonic outlet

In the case of the *subsonic outlet*, the static pressure is enforced on the outlet of a usually ducted flow. The rest of the flow variables are determined by the interior domain. The flow variables at the ghost cells are set as shown in (2.65).

$$\begin{bmatrix} \rho \\ u \\ v \\ w \\ P \end{bmatrix}_{Ghost} = \begin{bmatrix} \rho_{BN} \\ \mathrm{Ma}_{int} \cdot \sqrt{\frac{P_{ext} \cdot \gamma}{\rho}} \cdot \frac{u_{BN}}{\|\vec{U_{BN}}\|} \\ \mathrm{Ma}_{int} \cdot \sqrt{\frac{P_{ext} \cdot \gamma}{\rho}} \cdot \frac{v_{BN}}{\|\vec{U_{BN}}\|} \\ \mathrm{Ma}_{int} \cdot \sqrt{\frac{P_{ext} \cdot \gamma}{\rho}} \cdot \frac{w_{BN}}{\|\vec{U_{BN}}\|} \\ \frac{P_{ext}}{\rho_{BN}(\gamma-1)} + \frac{\|\vec{U_{BN}}\|}{2\rho_{BN}^2} \end{bmatrix} \tag{2.65}$$

In (2.65), Ma_{int} stands for the Mach number at the interior domain, P_{ext} is the exterior (enforced) static pressure and $\vec{U}$ the velocity vector $\vec{U} = [u, v, w]^T$.

2.7 Functionalities

This section describes various functionalities implemented in **mgOpt** by the author. Apart from their use in evaluating a function based on the flow solution, these are going to form the cost functions discussed later on in Chapter 3 and be an important part of the optimisation algorithm.

2.7.1 Lift and drag

The lift and drag forces acting on a solid boundary can be computed by first calculating the equivalent Cartesian force :

$$F_c = \oint P_t \cdot \vec{n} dS \tag{2.66}$$

where F_c the Cartesian force on the surface S, P_t the total pressure and $\vec{n}$ the normal to the surface vector. In the context of a finite volume, vertex centred solver, equation 2.66 takes the form of (2.67).

$$\vec{F_c} = \sum_{i=1}^{mB} P_t^i \cdot \vec{n_i} \cdot S_i \tag{2.67}$$

where mB the number of boundary faces, P_t^i the total pressure at the boundary node i amd $\vec{n}_i$ and S_i the normal to the boundary face normal and its area respectively. Once the Cartesian force $\vec{F_c}$ has been computed, the drag and lift forces can be evaluated as its components :

$$\|\vec{D}\| = \vec{F_c} \cdot \vec{V}_{flow} \tag{2.68}$$

$$\|\vec{L}\| = \sqrt{\|\vec{F_c}\|^2 - \|\vec{D}\|^2} \tag{2.69}$$

The lift and drag coefficients can also be computed, based onthe lift and drag values :

$$c_L = \frac{\|\vec{D}\|}{P_D \cdot A} = \frac{\vec{F_c} \cdot \vec{V}_{flow}}{P_D \cdot A} \tag{2.70}$$

$$c_D = \frac{\|\vec{L}\|}{P_D \cdot A} = \frac{\sqrt{\|\vec{F_c}\|^2 - \|\vec{D}\|^2}}{P_D \cdot A} \tag{2.71}$$

where $P_D = \frac{1}{2}\rho\gamma RT \cdot Ma^2$ the dynamic pressure (using the ideal gas low here, which is valid for air - ρ density, γ ratio of specific heats, R specific gas constant, T absolute temperature and Ma Mach number) and A the reference area.

2.7.2 Total pressure loss

A function of interest, especially for ducted flows, is the *total pressure loss*. It is computed by :

$$J = \int_{inlet} P_t dS - \int_{outlet} P_t dS \tag{2.72}$$

In the case of a vertex centred finite volume solver, like **mgOpt**, equation 2.72 takes the form :

$$J = \sum_{i=1}^{mB_{in}} P_t^i \cdot S_i - \sum_{i=1}^{mB_{out}} P_t^i \cdot S_i \tag{2.73}$$

where mB_{in} and mB_{out} the number of boundary faces at the inlet and outlet respectively, P_t^i and S_i the total pressure and the boundary face area of face i.

2.7.3 L^2_{norm} of total pressure

The difference between a computed pressure profile and a prescribed one can be quantified by the L^2_{norm} over wall. The mathematical formulation of such a function is given by :

$$J = \oint (P - P_{tar})^2 dS \tag{2.74}$$

where P the computed total pressure, P_{tar} the prescribed *target* pressure and S the surface. In the context of a finite volume solver, equation 2.74 becomes :

$$J = \sum_{i=1}^{mB} (P^i - P^i_{tar})^2 \cdot S_i \tag{2.75}$$

where mB the number of boundary faces and S_i the area of the i-th boundary face.

2.8 Convergence acceleration

2.8.1 Multi–grid

The multi–grid methodology [107] is an algorithmic technique, used to accelerate the convergence of the solution to a system of differential equations. This is achieved by using discretisations of varying resolutions for the same problem. The main operations taking place in a multi–grid algorithm are i. smoothing of fine discretisation oscillations, ii. solution and residual restriction from finer to coarser discretisation, iii. smoothing of coarse grid oscilations and iv. correction prolongation from coarser to finer discretisation. These operations can be performed in various algorithmic ways, e.g. from finest to coarsest and back to the finest (*V–cycle*) or with the addition of an intermediate smoothing on a semi–fine discretisation (*W–cycle*), Figure 2.9. The *V–cycle* is used throughout this research. Following any of the algorithms above, convergence acceleration is achieved by computing the numerical solution more on the (run–time cheaper) coarser and less on the (expensive) fine discretisation and by the removal of low frequency convergence oscillations, introduced by the fine discretisation. The methodology has been adopted by the CFD community, where it is typically met in three variants: *agglomeration* [72], *algebraic* [106] and *geometric* [30] multi–grid. Only the latter is used in the present research and shall be discussed in this section.

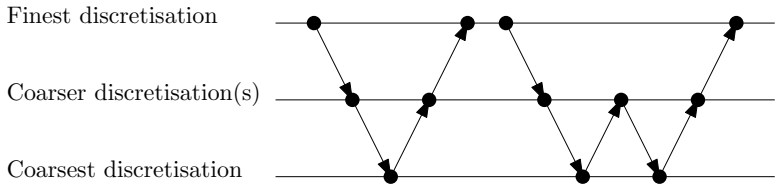


Figure 2.9: Examples of multi–grid cycles, the V [left] and the W [right] cycles.

The operations of restriction and prolongation between levels of varying discretisation resolution can be mathematically explained in the context of a CFD problem. The flow equations can be written in the form :

$$\mathcal{A}(U) = \Omega \tag{2.76}$$

where $\mathcal{A}$ is an operator on the solution, U the flow variables and Ω a right hand side. The exact solution $\hat{U}^h$ to the discretised system (2.76) on the finest level $[h]$ would satisfy equation :

$$\mathcal{A}^h(\hat{U}^h) = \Omega^h \tag{2.77}$$

Iteratively, the solution U is updated in every iteration in the form of :

$$U^{n+1} = U^n + \Lambda(\Omega - \mathcal{A}(U^n)) \tag{2.78}$$

where Λ stands for a time stepping algorithm. In every iteration, the error to the exact solution on the fine level would be given by :

$$E^h = \hat{U}^h - U^h \tag{2.79}$$

Therefore, in every iteration, (2.77) would be satisfied by :

$$\mathcal{A}^h(U^h + E^h) = \Omega^h \tag{2.80}$$

Subtracting $\mathcal{A}^h U^h$ from (2.80), a residual R^h is acquired for the fine level :

$$\begin{aligned} &\mathcal{A}^h(U^h + E^h) - \mathcal{A}^h U^h = \Omega^h - \mathcal{A}^h U^h \\ &\Rightarrow A^h(U^h + E^h) - \mathcal{A}^h U^h = R^h \end{aligned} \tag{2.81}$$

When the residual R^h is driven to zero, then the error E^h will be zero as well. Using the Full Approximation Storage (FAS) methodology [15], the restriction of the solution and the residuals from a fine to a coarser level would be performed in the next steps :

1. *Smooth errors on the finer level.*

$$U^n = U^n + \Lambda(\Omega^h - \mathcal{A}^h(U^n)) \tag{2.82}$$

2. *Transfer solution and residuals to the coarser level.*
   For this, geometrical transfer coefficients $I_h^{H,U}$ and $I_h^{H,R}$ are used for the solution and the residuals respectively, where $U^H = I_h^{H,U} U^h$ and $R^H = I_h^{H,R} R^h$. So, on the coarse level, (2.81) takes the form of (2.83).

$$\begin{aligned} &\mathcal{A}^H(U^H + E^H) - \mathcal{A}^H U^H = R^H \\ &\mathcal{A}^H(U^H + E^H) = \Omega^H \end{aligned} \tag{2.83}$$

   Therefore, the right hand side on the coarser level is $\Omega^H = R^H + \mathcal{A}^H U^H$. After, the solution and the residuals are transferred, the algorithm returns to *Step 1* until the coarsest level is reached. It shall be mentioned, that the number of smoothing iterations on each level may vary. In the results presented in this thesis, typically one smoothing iteration is used on all levels but the coarsest one, where the number

is usually higher (8-10 for two dimensional cases and 3-6 for three dimensional ones).

3. *Prolong the coarser level correction to the finer level.*
   After the coarser level is reached, smoothing is performed on it, equation (2.84).

$$U^H = U^H + \Lambda(\Omega^H - \mathcal{A}^H(U^H)) \tag{2.84}$$

   Then, the coarser level correction is prolonged to the finer level, equation (2.85), by using a prolongation operator I_H^h, which typically is the transpose of the residual restriction operator $I_H^h = (I_h^{H,R})^T$.

$$U^h = U^h + I_H^h(U^H - I_h^{H,U}U^h) \tag{2.85}$$

   This step is repeated until the finest level is reached.

The operations above describe a *V-Cycle* of multi–grid, Figure 2.9, but can be regrouped in any order to form another cycle, e.g. the *W-Cycle*.

In geometric multi–grid, the varying discretisation resolution is achieved by a group of decreasing density meshes, based on the same geometry. In this research, the legacy software *h!p* [77] is used for the generation of coarser meshes based on a fine one. This software was implemented in the past by former members of the research group and the supervisor of this thesis. Using the software for the present research, the author was able to frequently report bugs and propose alterations, improvements and additions for the tool, contributing to its further development.

h!p is using the edge/element–based algorithm of Moinier et al. [79] for the generation of geometric multi–grid. Its algorithm is based on two basic rules, which state that a set of edges can collapse only if the resulting geometry is still valid (negative volumes due to folded grids are not tolerated) and none of the neighbouring edges exceeds a certain multiple of its original length (design principle of the multi–grid: the length of edges is approximately doubled when coarsening). Based on those rules, the first step towards coarsening an isotropic mesh is to tag each edge with a maximum length, i.e. its length times a growth factor. Then, the elements are sorted in a heap list for smallest volume and the shortest edge and its conceptually parallel sibling are collapsed with a fixed maximum angle for the collapsed elements (so as to maintain a level of minimum mesh quality) and only if the rules allow it. Finally, a loop is run over all elements until there are no edges left to collapse. For meshes with stretched elements (e.g. in boundary layers) the algorithm is modified so as to perform only directional coarsening and prevent the long edges of the stretched layers to collapse. For this, all short edges in stretched areas are listed (chosen by a length–to–length ratio compared to longest

neighbouring edge and with the restriction that there is at least one other neighbouring edge of equivalent short length, pointing to the same direction). After this, coarsening is applied to all stretch regions but with no collapsing for neighbouring long edges to the ones to be collapsed. Once coarsening is finished on the stretched areas, the steps for isotropic grids is followed. In the end of this procedure, *H!P* provides the set of multi–grid meshes and their geometric coarse–to–fine connectivity, in terms of coarse element and contained fine mesh node. Examples of such meshes are presented in Figures 2.10 and 2.11 for a two and three dimensional case respectively.

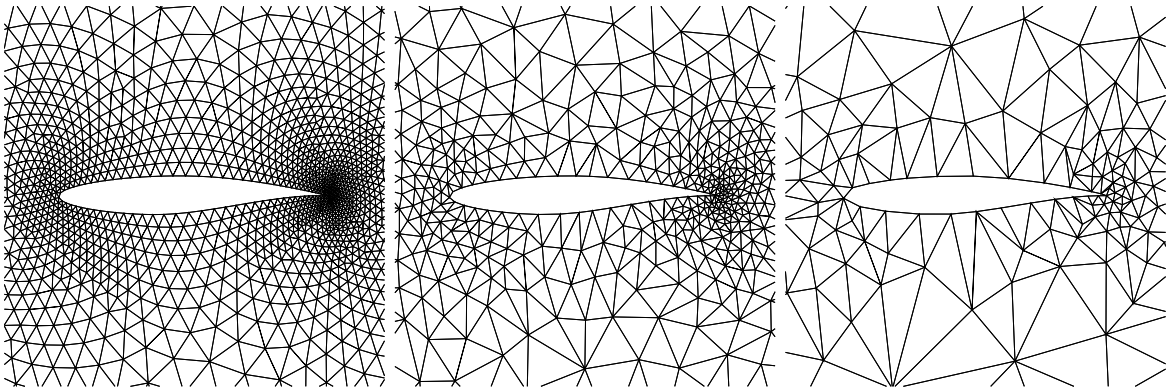


Figure 2.10: Geometric multi–grid meshes on a RAE 2822 airfoil, generated by *H!P*.

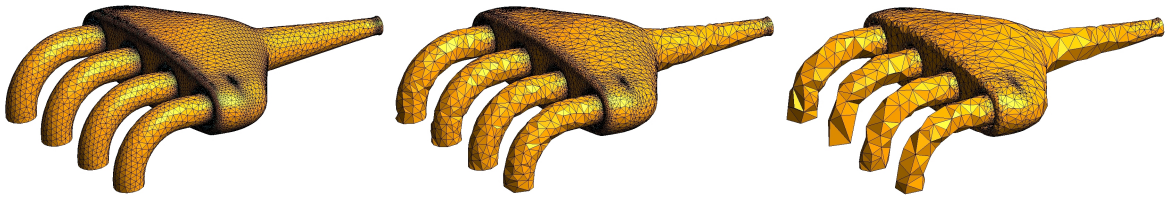


Figure 2.11: Geometric multi–grid meshes on a car engine part, generated by *H!P*

Provided a group of multi–grid meshes and the equivalent geometric connectivity, the restriction/prolongation operators can be computed. One approach to this would be by volume weighting. This logic existed in **mgOpt** for two dimensional meshes with triangular elements and was extended into three dimensions and tetrahedral elements by the author. According to this logic, the geometric transfer coefficients are computed by the formula :

$$I_{h,i}^{H} = \frac{\mathrm{V}_i^h}{\mathrm{V}_h} \tag{2.86}$$

where V_h is the volume of the coarse mesh element that contains the fine mesh node and V_i^h is the volume of the element formed by the fine mesh node, the coarse mesh node i and next node of the coarse mesh element, according to the elements orientation.

Once the version of **mgOpt**, which supported three dimensional cases, was prepared by the author, there was need for a more effective way for computing the transfer coefficients in hybrid meshes of potentially seven different element types (see Section 2.2). For this, a minimum–normal approach was implemented by the colleague Shenren Xu [109]. In this approach, the transfer coefficients are supposed to be the solution to the linear problem :

$$\begin{bmatrix} x_1 & x_2 & \cdots & x_n \\ y_1 & y_2 & \cdots & y_n \\ z_1 & z_2 & \cdots & z_n \\ 1 & 1 & \cdots & 1 \end{bmatrix} \begin{bmatrix} I^H_{h,1} \\ I^H_{h,2} \\ \vdots \\ I^H_{h,n} \end{bmatrix} = \begin{bmatrix} x_{FN} \\ y_{FN} \\ z_{FN} \\ 1 \end{bmatrix}$$

$$\Rightarrow \mathbf{A}\vec{I}^H_h = \vec{b}, \text{ with the constraint } I^H_{h,i} \geq 0 \tag{2.87}$$

where $\{x, y, z\}$ and FN represent coordinates and the fine mesh node respectively and n is the number of vertices forming the coarse mesh element. The problem is fully determined by arguing that the solution to the system is the one that minimises the Euclidean norm of the transfer coefficients vector $\|\vec{I}^H_h\| = \sqrt{\sum_{i=1}^{n}(I^H_{h,i})^2}$. One way to compute the solution $\vec{I}^H_h$ to this minimum norm problem is by using reduced QR decomposition [100] for the rectangular ($[m \times n], m > n$) matrix $\mathbf{A}^T = \mathbf{QR}$, where $\mathbf{Q}$ an $[m \times m]$ unitary matrix and $\mathbf{R}$ an $[m \times n]$ upper triangular matrix :

$$\begin{aligned} \mathbf{A}\vec{I}^H_h = \vec{b} \Rightarrow \vec{I}^H_h &= \mathbf{A}^{-1}\vec{b} \\ &= (\mathbf{A}\mathbf{A}^T\mathbf{A}^{-T})^{-1}\vec{b} \\ &= \mathbf{A}^T(\mathbf{A}\mathbf{A}^T)^{-1}\vec{b} \\ &= \mathbf{QR}(\mathbf{R}^T\mathbf{Q}^T\mathbf{QR})^{-1}\vec{b} \\ \Rightarrow \vec{I}^H_h &= (\mathbf{QR})^{-T}\vec{b} \end{aligned} \tag{2.88}$$

Using this approach, the algorithm built to solve the entire problem (with the positivity constraint) follows the steps: i. assemble the matrix $\mathbf{A}$ and the right hand side term $\vec{b}$, ii. compute the $\mathbf{QR}$ decomposition of $\mathbf{A}^T$, iii. compute the unconstrained minimum norm solution via 2.88 for all $i \in [1, ..., n]$ and iv. if $I^H_{h,i} < 0$ remove node i and go to step [i] with the remaining nodes or, if $I^H_{h,i} = 0 \ \forall i$, output $I^H_{h,i}$.

The above result in convergence acceleration. This can be demonstrated on the NACA 0012 case of Figure 2.12. The results presented are for second order accurate inviscid flow and four levels of multi–grid (4400 to 28 elements). Ten smoothing iterations were performed on the coarsest level and one on the rest. Similar behaviour can be observed the three dimensional case of second order accurate laminar flow through an S-Bend, Figure 2.13. Four levels were used (173000 to 3300 elements) and two smoothing iterations were

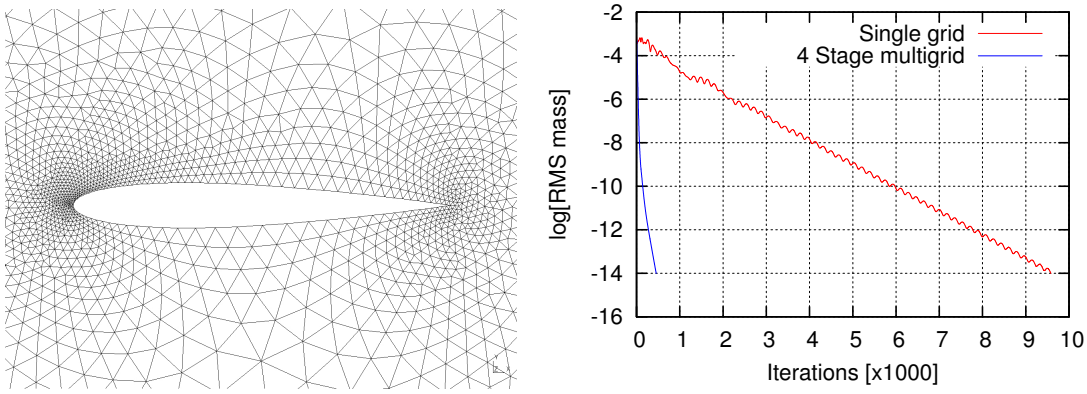


Figure 2.12: Multi–grid converge acceleration on a NACA 0012 airfoil: finest mesh [left] and convergence comparison [right]

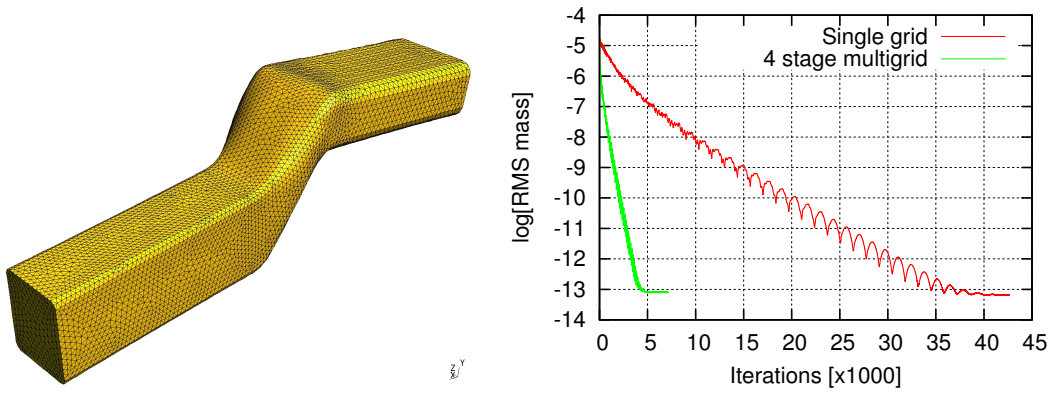


Figure 2.13: Multi–grid converge acceleration for an S-Bend: finest mesh [left] and convergence comparison [right]

performed on the coarsest level and one on the rest.

2.8.2 Pre–conditioning

Further convergence acceleration can be achieved by pre–conditioning the system of flow equations. Especially in the case of viscous flow, a pre-conditioner is necessary in order to maintain run–times to acceptable levels. For this reason, the block Jacobi pre–conditioner [3] has been implemented for simultaneous use with multi–grid [90, 79] by other members of the research group [109]. In the context of this thesis, the author used this pre–conditioner on the adjoint system, work that is going to be presented in Section 3.6.3. For steady flows with varying time–step Δt_i for each control volume i, the block Jacobi pre–conditioner aims to accelerate convergence by taking into account the different speeds,

with which the flow variables propagate. In this way, a different time step is applied to each of the variables. Using Newton's method and an intermediate solution Q_i^n, the solution would be updated as :

$$Q_i^{n+1} = Q_i^n - \left[\frac{\partial R_i}{\partial Q_i}(Q^n)\right]^{-1} R_i^n(Q^n) \tag{2.89}$$

where the nomenclature of Section 2.5 is used. Therefore, the block Jacobi pre–conditioner takes the form :

$$\mathcal{B}_i = \Delta t_i \begin{bmatrix} \frac{\partial R_1}{\partial Q_1}(Q^n) & 0 & 0 & 0 & 0 \\ 0 & \frac{\partial R_2}{\partial Q_2}(Q^n) & 0 & 0 & 0 \\ 0 & 0 & \frac{\partial R_3}{\partial Q_3}(Q^n) & 0 & 0 \\ 0 & 0 & 0 & \frac{\partial R_4}{\partial Q_4}(Q^n) & 0 \\ 0 & 0 & 0 & 0 & \frac{\partial R_5}{\partial Q_5}(Q^n) \end{bmatrix}^{-1} \tag{2.90}$$

and the flow solution can now be updated as :

$$Q_i^{n+1} = Q_i^n - \mathcal{B}_i R_i^n(Q^n) \tag{2.91}$$

Once implemented, the block Jacobi pre–conditioner offers an even further convergence acceleration in the case of viscous flows. This is demonstrated in Figure 2.14, where the viscous flow over a flat plate is examined.

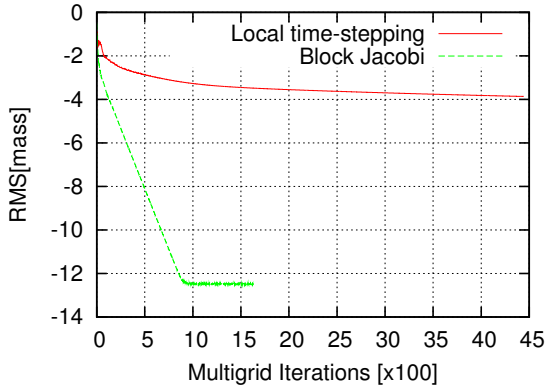


Figure 2.14: Convergence acceleration using the block Jacobi pre–conditioner

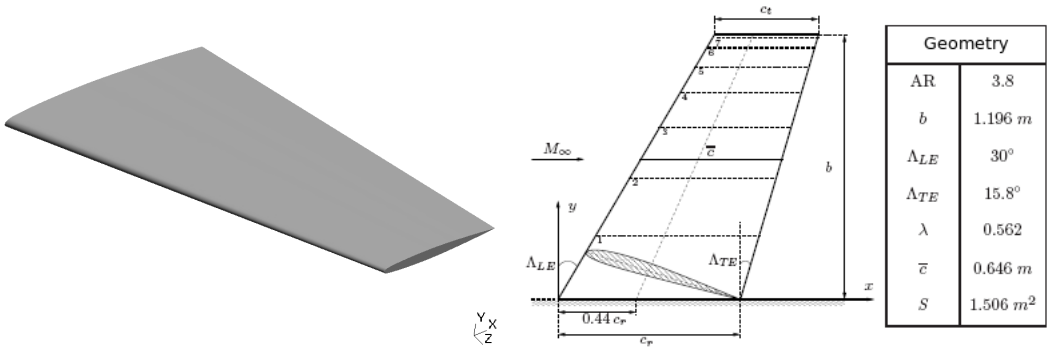
2.9 Flow solver validation and results

The next paragraphs present a number of benchmark literature cases, used for the validation of the flow solver of **mgOpt**. Apart from those, a comparison is also made

between the latter and the commercial CFD solver Fluent [4] for a non-benchmark case. These will prove that the flow solver is trustworthy and therefore its differentiation and derivation of the adjoint solver from it does make sense.

2.9.1 Inviscid transonic flow over an ONERA M6 wing

The transonic ONERA M6 wing (Figure 2.15) was used to validate the inviscid part of the flow solver. It was chosen because there are detailed wind tunnel experimental data [81] to compare against. The second order accurate Euler results should be qualitative comparable to the experimental data for this simple configuration. It should be mentioned that this validation case was assigned to the student Mateusz Gugala as a part of his masters project.



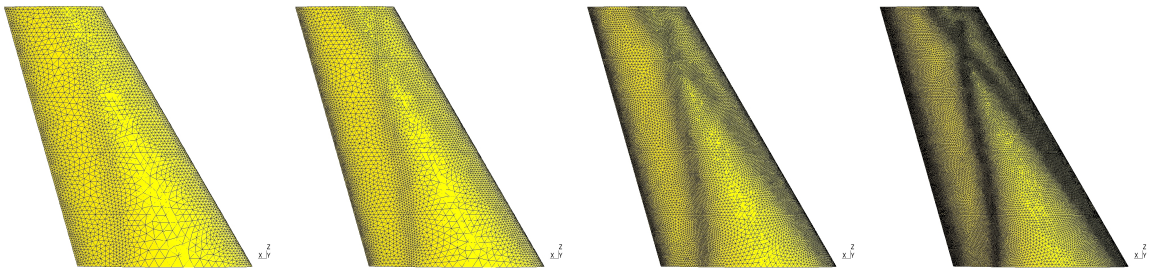
Geometry	
AR	3.8
b	1.196 m
Λ_{LE}	30°
Λ_{TE}	15.8°
λ	0.562
$\bar{c}$	0.646 m
S	1.506 m^2

Figure 2.15: ONERA M6 wing [81]. CAD model prepared in Siemens NX6 [left] and geometrical data [right].

The case was at the same time examined as a mesh convergence study, using four different meshes (Figure 2.16). The free–stream conditions were Mach number $Ma = 0.8395$ and angle of attack $\alpha = 3.06°$. Solutions were acquired on all meshes for second order accuracy, using both Roe's (2.3.1) and $AUSM^+_{up}$ (2.3.2) flux functions and Venkatakrisnan's limiter (2.4.2). The boundary conditions were set as *slip wall* (2.6.2) on the wing and the symmetry plane and as *farfield* (2.6.1) on the outer domain, Figure 2.17.

The experimentally computed lift coefficient in the wing tunnel at NASA [81] was measured to be 0.26. In Table 2.2, the computed lift coefficients for the meshes and settings above are presented.

Comparison against the experimental data was also performed for the pressure distribution at sections one to six (Figure 2.15). The results are summarised in Figure 2.19 and 2.20. These figures show a close match between the results computed by **mgOpt** and the experimental measurements, even for these inviscid computations. Therefore, the inviscid part of **mgOpt** is considered valid for both available flux functions.



(a) Mesh 1 (b) Mesh 2 (c) Mesh 3 (d) Mesh 4

Figure 2.16: Various surface meshes on the ONERA M6 wing. All the meshes consist of only tetrahedral elements and have been refined in the shock areas.

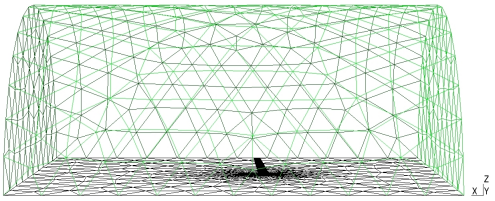


Figure 2.17: Imposed boundary conditions on the ONERA M6 wing: *slip wall* [black - wing and symmetry plane] and *farfield* [green - outer domain].

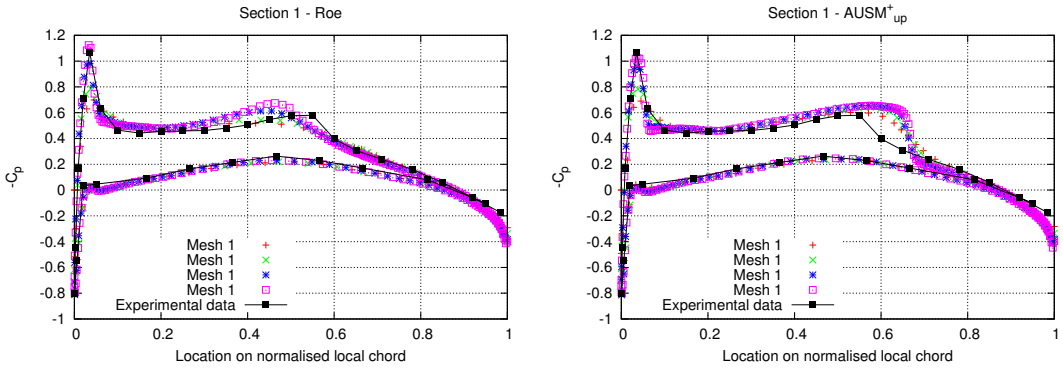


Figure 2.18: Pressure coefficient distributions on various ONERA M6 wing sections, part one.

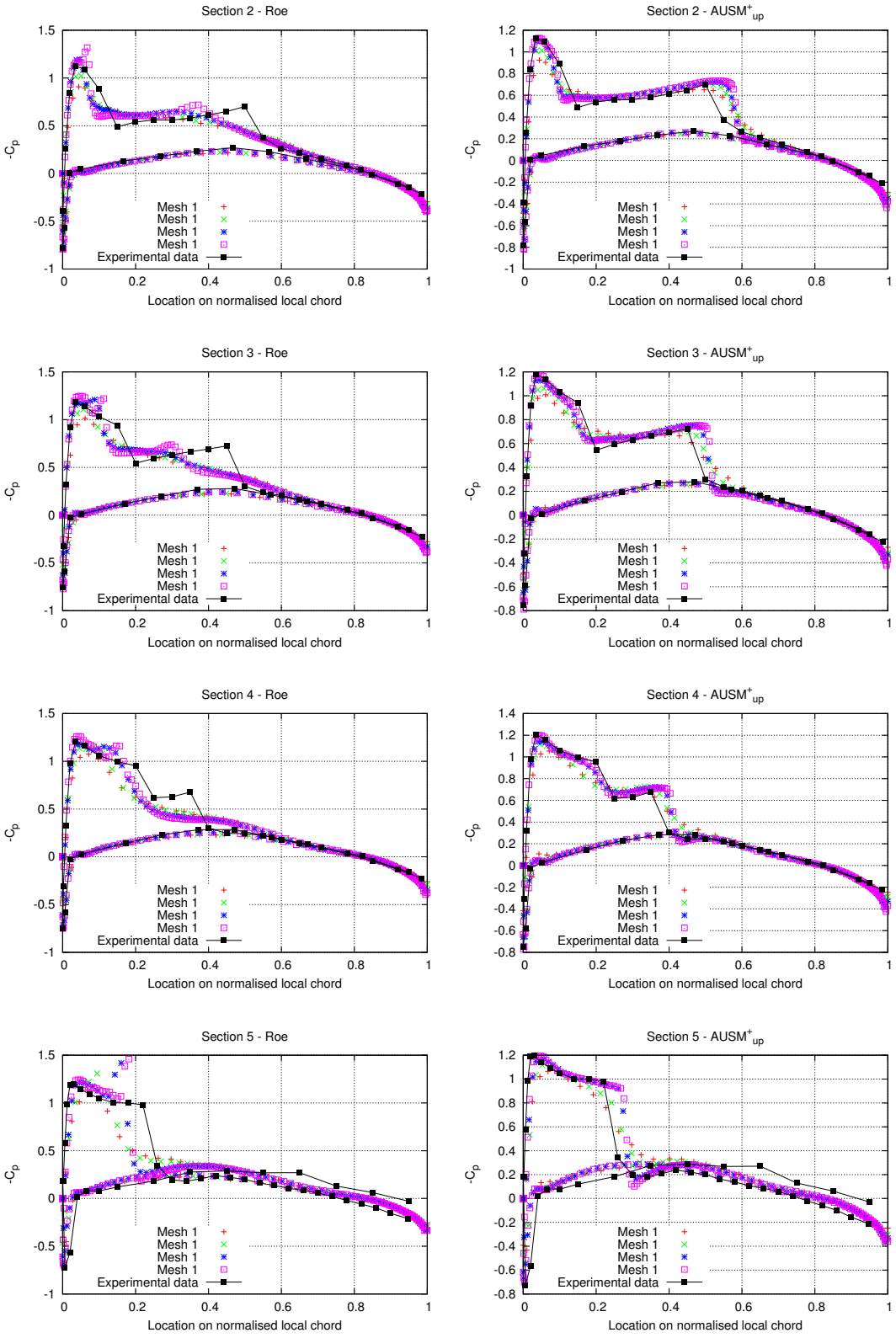


Figure 2.19: Pressure coefficient distributions on various ONERA M6 wing sections, part two.

Mesh	Elements	Lift Coefficient (Roe)	Lift Coefficient ($AUSM^{+}_{up}$)
Mesh 1	158655	0.256	0.277
Mesh 2	272831	0.263	0.282
Mesh 3	715235	0.268	0.285
Mesh 4	1288695	0.271	0.289

Table 2.2: Mesh characteristics and computed lift coefficients on the ONERA M6 wing. The experimental value of the lift coefficient is $c_L = 0.26$.

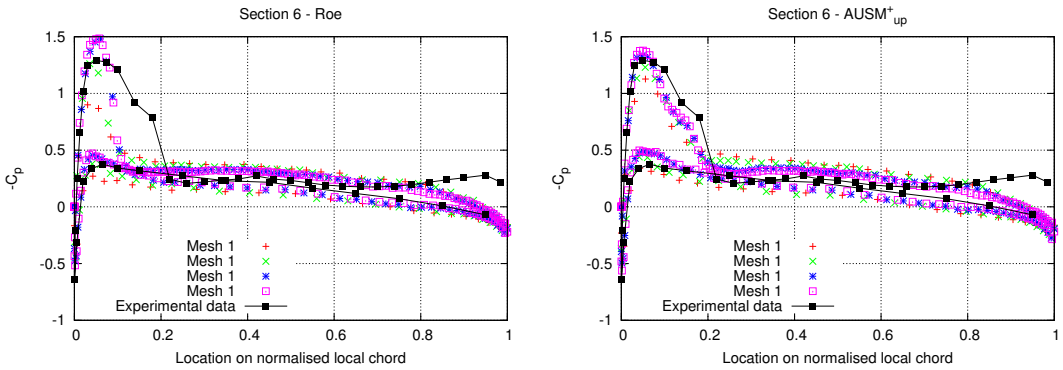


Figure 2.20: Pressure coefficient distributions on various ONERA M6 wing sections, part three.

A visual result of the solution can be observed in Figure 2.21, where the velocity and total pressure fields are presented.

2.9.2 Viscous flow over a flat plate

The viscous part of **mgOpt** was validated against the robust vertex–centred flow solver *FUN3D* [80] of NASA. The configuration was the three dimensional flat plate of Figure 2.22. Detailed results on the latter, using the Spalart–Allmaras turbulence model

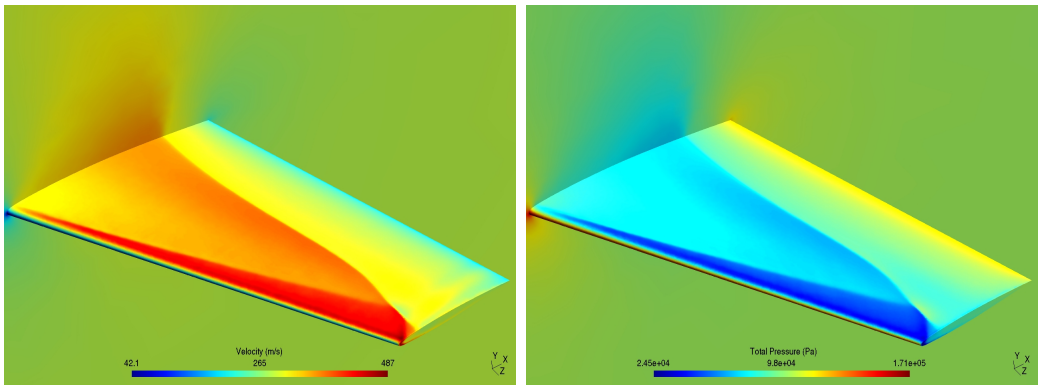


Figure 2.21: Velocity [left] and total pressure [right] on the ONERA M6 wing.

in *FUN3D* (same with **mgOpt**), are available as a benchmark case [82]. It shall be mentioned that the following results were computed by the colleague Shenren Xu.

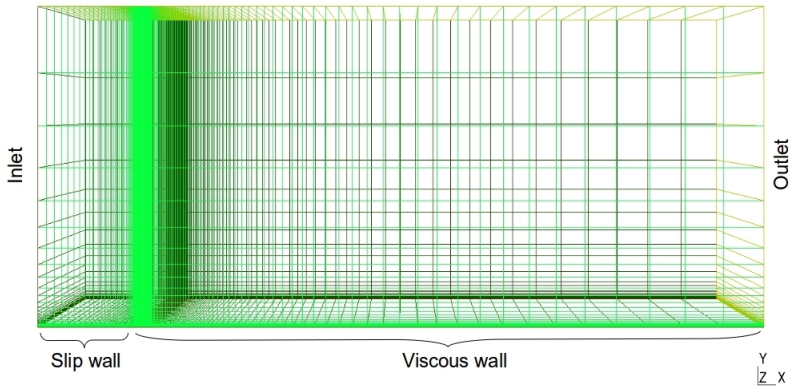


Figure 2.22: Description of the viscous flat plate problem.

The set-up was for inlet Mach number $Ma = 0.2$ and Reynolds number $Re = 5000000$ (for unit characteristic length). The finest mesh used in **mgOpt** was consisting of 37050 hexahedra. The quantities of most interest in this case were the surface skin friction coefficient c_f, equation (2.92), and the shear velocity $U^+ = \sqrt{\tau_w/\rho}$.

$$c_f = \frac{\tau_w}{\frac{1}{2}\rho U_\infty^2} \tag{2.92}$$

Figure 2.23 presents the comparison between **mgOpt** and the benchmark data. It can be observed that the results are nearly identical. Therefore, the viscous part of **mgOpt** is considered to be valid as well.

2.9.3 Laminar flow through an S-Bend duct

To conclude the series of flow results, an internal laminar flow case is examined, through a bended duct. The geometry was provided by Volkswagen and is an air acclimatisation duct of the VW Golf. The geometry and an equivalent mesh are shown in Figure 2.24. The geometry has been prepared in the CAD package *CATIA* [26] and the meshing was carried out in Ansys *GAMBIT* [5]. In this case, the surrounding duct is a *No-slip Wall* (section 2.65), the inlet a *subsonic inlet* (section 2.6.4) and the outlet a *subsonic outlet* (section 2.6.5). The inlet velocity is 30m/s and the boundary conditions set as no-slip wall and subsonic inlet/outlet.s

2.9.4 Viscous flow over a passenger car

As a last demonstration of the flow solver, a case of viscous flow over a passenger car is examined. The car is courtesy of Volkswagen and it is the Passat model. The case was

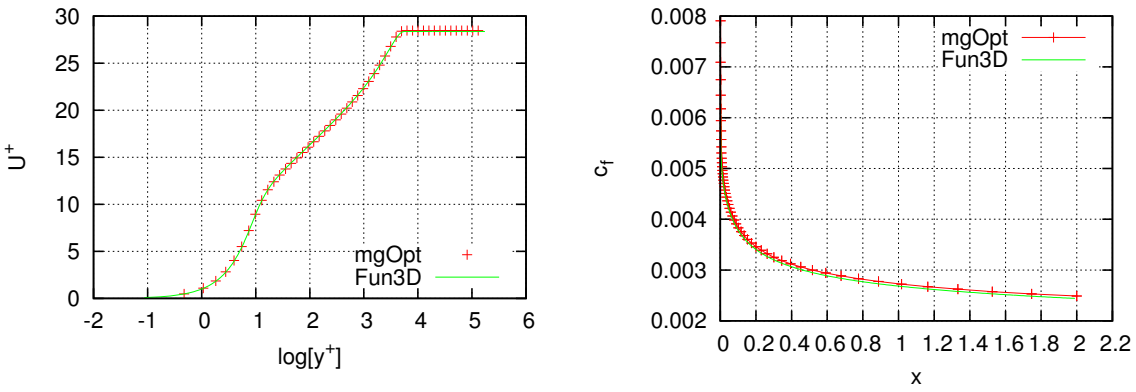


Figure 2.23: U^+ profile [left] and skin friction [right] on the viscous flat plate benchmark case of NASA [82]. The results of **mgOpt** match and *FUN3D* [80] are nearly identical.

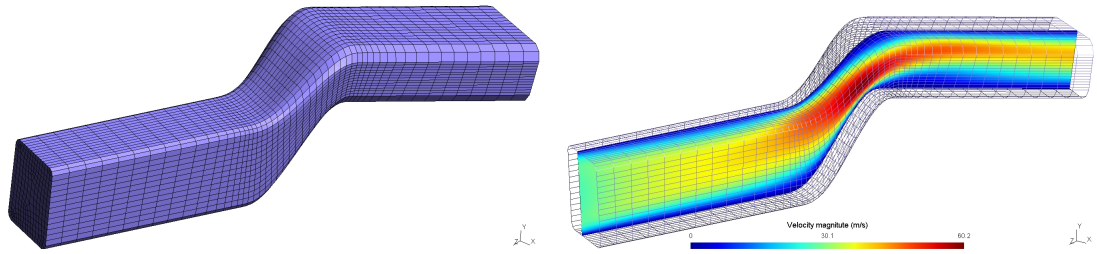


Figure 2.24: Geometry and discretisation [left] and flow [right] through an S-Bend duct.

run for a speed of 120km/h and Figure 2.25 presents the pressure map on the car. This case is going to be used in the next chapter for demonstration of design vectors.

2.10 Summary

In this chapter, the flow solver that was further developed during this study has been presented. The spatial and temporal discretisation and flux functions implemented have been presented. Apart from these, solution accuracy improvement methods have been discussed and the boundary conditions used have been described. In order to validate and demonstrate the capabilities of the flow solver of the current version of **mgOpt**, a variety of external and internal flow cases were presented. The analysis of these solutions shows that the flow solver is capable of solving a variety of fluid flow problems.

Closing this chapter and in contrast with other CFD studies, the flow is not going to be the main focus in the following chapters. The flow solver has been developed only in order to form the basis of for the derivation of sensitivity solvers, such as the adjoint (Chapter 3), tangent (Chapter 3) and Hessian (Chapter 5). It is within the

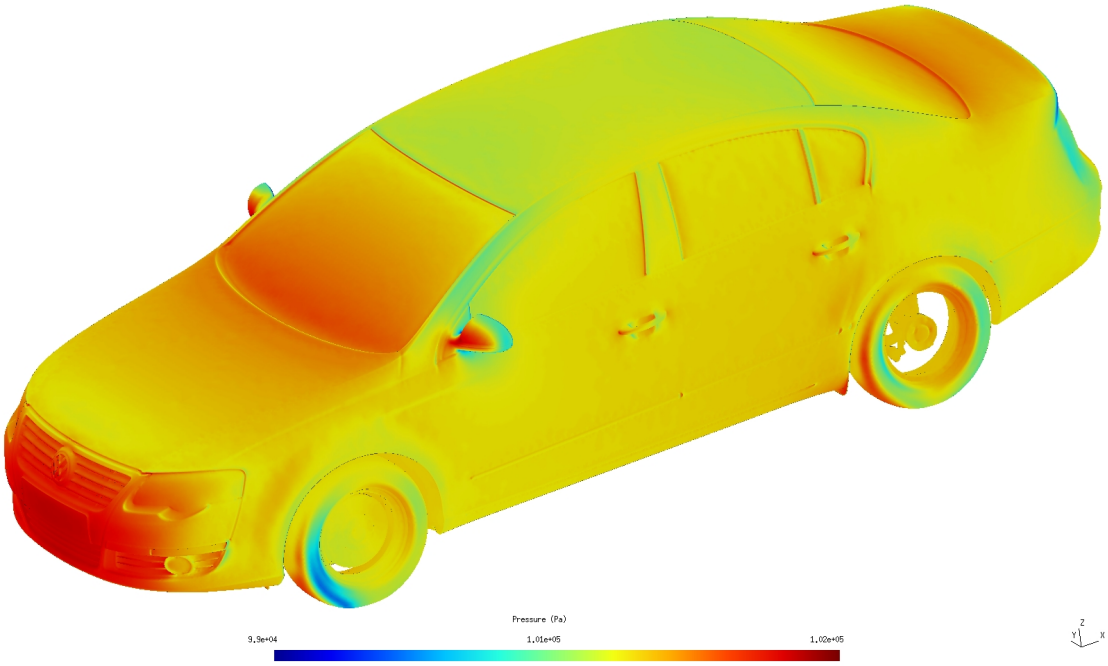


Figure 2.25: Pressure contours on the VW Passat.

author's knowledge that a lot more could be done so as to improve and extend the capabilities of the flow solver, but this is of secondary importance to this research. As it will be demonstrated in the next chapters, whichever improvement/addition is made to the flow solver, can be automatically incorporated in the adjoint, tangent, Hessian and generally anything that the optimisation algorithm incorporates, via the use of Automatic Differentiation and advanced scripting techniques.

Chapter 3

The adjoint solver

3.1 Shape optimisation and adjoints

In aerodynamic shape optimisation problems, there are three main components describing the problem : the *Design Variables* $[\alpha]$, the *Flow Variables* $[Q]$ and the *Cost Function(s)* $[J]$. The design variables determine the shape of a solid surface. Such examples can be the control points of B-Splines, Non-Uniform Rational B-Splines (NURBS) etc. or even the nodes of a discretised surface themselves (node–based parametrisation), Figure 3.1. In order to construct a CFD problem, which will be used for optimisation later on, the following relations are considered: i. the design variables α determine the design surface S, ii. out of the surface S the surface discretisation X is determined, iii. the surface discretisation X determines the space discretisation V, iv. the flow variables U are computed using the space discretisation V and finally iv. the cost function J is dependent on the flow and design variables α. The relations above imply a dependency of the cost function J on the rest of the involved parameters. This dependency can be summarised as :

$$J = J(U(V(X), X(S(\alpha)))), \alpha) \tag{3.1}$$

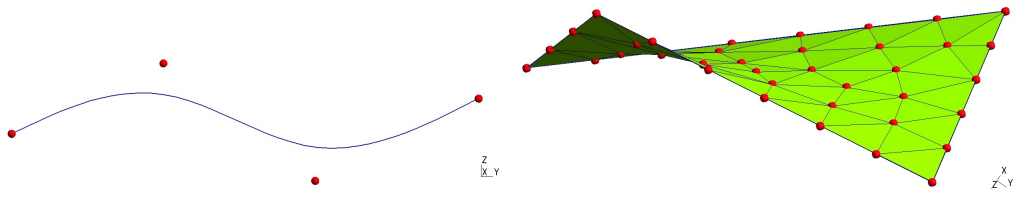


Figure 3.1: Examples of design variables' parametrisation, B-Splines [left], node-based parametrisation [right]. The design variables are marked with red spheres.

The next step towards a gradient based optimisation algorithm is the computation of the sensitivity of the cost function J with respect to the design variables $\vec{\alpha}$. There are different ways, in which this information can be acquired, such as *Finite Differences* [50, 94], *Direct Differentiation* and the *adjoint* approach. The implementation of Finite Differences (FD) is rather simple but their computational runtime cost increases linearly with the number of design variables (at a rate of two or three for forward/backward or central scheme respectively). So, for realistic optimisation cases with of hundreds of design variables, the use FD for gradient computation soon becomes prohibitive. Also, the perturbation step δx used for FD determines the levels of round-off and truncation errors. A case–dependent value for δx has to be determined so that either do not dominate. This makes this approach and the equivalent computed gradients very dependent on δx. For the reasons above, FD is not the most suitable methodology for optimisation. Instead, direct differentiation or the adjoint approach may be used. These will be discussed in the following sections.

3.2 Direct differentiation

The logic of direct differentiation is to apply the chain rule to (3.1). Supposing that R are the residuals of the partial differential equations characterising the flow problem, the solution of the latter can be represented as :

$$R(U, \alpha) = 0 \tag{3.2}$$

U are the flow variables and α the design parameters. The derivative of a cost function $J = J(U, \alpha)$ with respect to each design variable α can be acquired by applying the chain rule :

$$\frac{dJ}{d\alpha} = \frac{\partial J}{\partial \alpha} + \frac{\partial J}{\partial U}\frac{dU}{d\alpha} \tag{3.3}$$

The most expensive term to compute is the sensitivity $dU/d\alpha_i$, which is the sensitivity of the flow variables with respect to the design variables. This is calculated by differentiating the (3.2) with respect to α :

$$\begin{aligned} \frac{d}{d\alpha}\left(R\left(U, \alpha\right)\right) &= 0 \\ \Rightarrow \frac{\partial R}{\partial U}\frac{dU}{d\alpha} &= -\frac{\partial R}{\partial \alpha} \\ \Rightarrow \mathbf{A}u &= f \end{aligned} \tag{3.4}$$

where $\mathbf{A}$ is the flux Jacobian (derivatives of the fluxes with respect to the state variables), $u = \partial U/\partial\alpha$ and $f = -\partial R/\partial\alpha$ a source term, which is added to the conservation equations by the shape change. Although direct differentiation is more accurate and robust methodology to compute gradients compared to FD, the cost of solving the expensive system (3.3) still scales linearly with the number of design variables. This scaling is at factor of one, so less than FD, but still prohibitive for problems with large number of design variables. This problem can be dealt with, when the above computations become independent of the number of design variables, by using the adjoint approach. This will be discussed in the following section.

3.3 The adjoint approach

The adjoint methodology for the computation of gradients is characterised by the independence of the number of design variables, as it will be explained in this paragraph. This makes this approach the most suitable for gradient based optimisation problems with a large number of design parameters. It was first used in aerodynamic shape optimisation by Pironneau [91] and Jameson [51] and triggered interest for future research, both in academia and industry. The two main approaches to derive an adjoint system are the *continuous* [51, 91] and the *discrete* [38, 74] approach. Any of the two methodologies may be used to derive the adjoint system by hand and implement the equivalent sensitivity code. The discrete adjoint sensitivity code though can also be acquired automatically with the use of Automatic Differentiation (AD). The present research aims to further explore the use of AD for this purpose and contribute to the existing literature in the field and therefore, the **discrete** adjoint approach is the one that is examined.

Equation (3.3) implies that the computation of $dJ/d\alpha$ involves the solution of the expensive system (3.4) for every design variable α_i. The equations are repeated here for convenience:

$$\frac{dJ}{d\alpha} = \frac{\partial J}{\partial \alpha} + \frac{\partial J}{\partial U}\frac{dU}{d\alpha} \Rightarrow \frac{dJ}{d\alpha} = \frac{\partial J}{\partial \alpha} + g^T u \tag{3.3}$$

$$\frac{\partial R}{\partial U}\frac{\partial U}{\partial \alpha} + \frac{\partial R}{\partial \alpha} = 0 \Rightarrow \mathbf{A}u = f \tag{3.4}$$

The system of adjoint equations is formed to be similar to (3.4), but using the transpose of the Jacobian matrix $\mathbf{A}$ and the transpose of the derivative of the cost function J with respect to the state variables U, $g = \frac{\partial J}{\partial U}^T$, as right hand side. The solution to this

equation, is the *adjoint* solution :

$$\begin{aligned} \frac{\partial R}{\partial U}^T v &= \frac{\partial J}{\partial U}^T \\ \left(\Rightarrow \frac{\partial R}{\partial U}^T \frac{dJ}{dR}^T \right. &= \left. \frac{\partial J}{\partial U}^T \right) \\ \Rightarrow \mathbf{A}^T v &= g \end{aligned} \tag{3.5}$$

where v shall be referred to as the *adjoint variables*. The system above is independent of the of design parameters α and, as a result, it has to be solved only once. Using (3.5) and (3.4), the term $g^T u$ of (3.3) becomes :

$$\begin{aligned} g^T u = \left(\mathbf{A}^T v\right)^T u &= u^T \mathbf{A} u = v^T f \\ \Rightarrow g^T u &= v^T f \end{aligned} \tag{3.6}$$

Equation (3.6) is an important equation in understanding the adjoint methodology and it will be referred to as *adjoint–direct differentiation equivalence*. Using (3.6), (3.3) can be written as :

$$\frac{dJ}{d\alpha} = \frac{\partial J}{\partial \alpha} + v^T f \tag{3.7}$$

Hence, the derivative of the cost function with respect to the design variables can be computed either using (3.3) or (3.7). In comparison to the disadvantages of using (3.3) discussed in Section 3.2, the adjoint variables v (or *dual*) depend only on the transposed Jacobian $\mathbf{A}^T$ and the choice of the cost function g, meaning that they can be computed once for all design variables. So, the cost for the computation of the gradient $dJ/d\alpha_i$ becomes independent of the number of design variables (as opposed to the tangent–linear model, where this computation scales with the number of design variables).

A more efficient computation of the gradient of the cost function can be achieved by considering the transpose of equation (3.7) :

$$\begin{aligned} \frac{dJ}{d\alpha}^T &= \left(\frac{\partial J}{\partial \alpha} + v^T f \right)^T \\ \Rightarrow \frac{dJ}{d\alpha}^T &= \frac{\partial J}{\partial \alpha}^T + f^T v \\ \Rightarrow \frac{dJ}{d\alpha}^T &= \frac{\partial J}{\partial \alpha}^T + \frac{\partial R}{\partial \alpha}^T v \end{aligned} \tag{3.8}$$

To compute the partial derivative $\frac{\partial R}{\partial \alpha}$, the direct dependency of the residuals R of the partial differential equations describing the flow to the design variables α must be con-

sidered. Using the nomenclature of equation (3.1), this dependency can be summarised as :

$$R = R(V(X(S(\alpha)))) \tag{3.9}$$

Therefore, the source term f can be written as :

$$f = \frac{\partial R}{\partial \alpha} = \frac{\partial R}{\partial V}\frac{\partial V}{\partial X}\frac{\partial X}{\partial S}\frac{\partial S}{\partial \alpha} \tag{3.10}$$

Equation (3.8) could now take the form :

$$\frac{dJ}{d\alpha}^T = \frac{\partial J}{\partial \alpha}^T + \frac{\partial S}{\partial \alpha}^T \frac{\partial X}{\partial S}^T \frac{\partial V}{\partial X}^T \frac{\partial R}{\partial V}^T v \tag{3.11}$$

By computing the gradients of the cost function in the transposed manner of the last equation, any dependency of the design variables is moved to the end of the multiplication chain (term $\frac{\partial S}{\partial \alpha}^T$). So, the term $\beta = \frac{\partial X}{\partial S}^T \frac{\partial V}{\partial X}^T \frac{\partial R}{\partial V}^T v$ is independent of the design variables and can be computed only once. This reduces computational time further and makes the gradient computation more efficient. The logic of equation (3.8) was adopted by the author in **mgOpt**.

3.4 Discrete adjoint via Automatic Differentiation

Apart from the classic mathematical approach (see previous paragraphs), the gradients of a function can be also derived by the methodology of algorithmic differentiation [46]. This methodology breaks the function into elementary operations, differentiates those and then assemble the elemental derivatives into the gradient of the function, based on the chain rule. This can be applied in both direct differentiation and adjoint manner, namely ***forward*** and ***reverse*** accumulation respectively in terms of algorithmic differentiation. This methodology generates lots of interest in the scientific community, especially in the field of mathematics and computer science, the main reasons being the opportunity to automatically derive machine accurate gradients (see [46]) of functions implemented in a programming language. In the past years, a number of software tools have been developed for this purpose, supporting a variety of programming languages and resulting into the so–called **Automatic Differentiation** (AD). An example of automatic differentiation and use of AD is given in Appendix A.

There are two main approaches to AD, *operator overloading* (OO) and *source code transformation* (SCT) [99, 69]. The first is applied by overloading objects of elementary mathematical operations and real numbers. In practice, AD following this methodology can be used on source codes written in a programming language that supports operator

overloading, such as Fortran 90/95, C++, Python etc. In the second case of SCT, the AD tool automatically parses the supplied source code and generates the equivalent derivative source code. The logic of SCT can be used on any programming language, although the development of the equivalent AD tool is more laborious than that of OO. There is a number of AD tools featuring both methodologies, an up–to–date list of which can be found at *www.autodiff.org*. For the present research, SCT was used, the main reason being that SCT allows user to view and read of the derivative source code and decide upon parts that can be optimised to promote performance. The aim was not to use AD as a far–end user but to investigate improvements, report to the AD community and exploit the optimal use of AD on a CFD code written in an modern programming language. The tool *Tapenade* [101] is used throughout this thesis.

Moving on to the use of AD for the generation of adjoint CFD codes, despite what the name might imply, the process is actually not "automatic" in most cases. There as many issues that have to be dealt with until the process becomes fully automatic. This is also the reason why the industry has not adopted the methodology yet, as there is lots of labour involved in the equivalent process, without guarantee of the result. This thesis addresses those issues and presents a state of the art use of AD to automatically generate sensitivity code. For this, high level object oriented Fortran 90/95 was used, which incorporates derived data types, pointers, modules and all the high level features of the language. All the issues and the ways to bypass them are discussed in the next sections.

3.4.1 Source code preparation

The source code of a program, in this case a CFD solver, contains various functionalities, only a fraction of which needs to be passed to an AD tool for differentiation. Once those parts are identified, they could theoretically be passed to the tool and derive the equivalent sensitivity code. To determine those, equations (3.11) and (3.5) need to be examined. These involve differentiation of the parts of the source code that compute geometric quantities (volumes, normals, etc.), the flow and the cost function. Two issues might arise when submitting these parts to AD. First, these files may contain language features that cannot be parsed by the tool. The development of AD tools is still on going and not all aspects of modern languages are supported. For example, *Tapenade* cannot identify dependencies within derived data types in reverse mode. This can be dealt with by adjusting the source code to be compatible with the tool's capabilities. This could be a time consuming process, especially in the case of large industrial codes, that requires a good knowledge of what is supported. This may not even be enough, as

there might be features that have not been exploited by the AD tool developers yet. For example, while differentiating **mgOpt**, the author was reporting undocumented bugs and unsupported elements to the developers of *Tapenade*, helping towards its improvement. Second, functions within the files supplied to the tool might cause problems due to their high level syntax. Although these functions are not differentiated, they are still parsed by the tool, which might not be able to understand their programming. Such functions need to be stripped out from those files, rather than going through the laborious task of readjusting the code. To achieve this automatically, the source code can be preprocessed with the text/macro preprocessor (e.g. *GNU C* [39] or *GNU M4* [42]), process which is controlled via a *Makefile*. An example of this, applied on **mgOpt**, is given in Figure 3.2, where introduced preprocessing pragmas will remove the code that is not to be seen by the AD tool code.

```
#ifndef REMOVE_IN_AD

!123456789 123456789 123456789 123456789 123456789 123456789 123456789 12
!
! quick_sort_arr2
! Quicksort on an array list(2,*), eg a list of edges

  recursive subroutine quick_sort_arr2 (list, mBEg)

        ...

  end subroutine quick_sort_arr2

#endif
```

Figure 3.2: Preprocessing pragmas in the source code. The subroutine will be removed from the file before the file that contains it is supplied to the AD tool.

After these two basic steps, the source code will be ready for differentiation. Before moving to the next paragraph, which describes an effective way of using the AD tool, it shall be mentioned that the biggest obstacle in the preparation of the code is recoding. Although the AD tools have evolved a lot and are very capable today, there are still issues that have not been dealt with. Therefore, rearranging the source code is still a non-automatic step in the procedure, which might put the code developers off using AD. It can require lots of effort until the source code can be differentiated, a cost that grows with the scale of the latter. In the case of large industrial codes, which are not written with the aim of using AD, this may even be prohibitive. As far as **mgOpt** is concerned, the coding strategy was to use as complex coding as possible, push the AD tool to its syntax limits and propose changes and improvements. The relative work also contributed to the application of AD on the industrial code ACE+ of ESI by other members of the research group, which to the author's knowledge is the first application of AD on a commercial CFD solver written in a high level language.

3.4.2 Application of AD

Once the source code has been prepared, the AD tool will be able to parse it and generate the sensitivity code. The call to the tool is a process that can be automated. In the context of this thesis, this is performed by augmenting the *Makefile*, so that the generation of the sensitivity code is similar to the compilation of a program. In this way, the program can be chosen to include sensitivity code or not at compile time. This shows the potential of using AD. Even if someone does not have knowledge on sensitivity codes, they can generate and use it in just a single line call. It would make no difference to the end user compared to simple instructions on compilation. Such a property is essential for large codes involving a number of developers and provides the highest level of code maintenance possible. An example of the methodology is given in Figure 3.3, which includes rules and call to the AD tool *Tapenade*. It can be observed that, the rules include instructions for both forward and reverse mode, any of which can be selected to be included in the final executable.

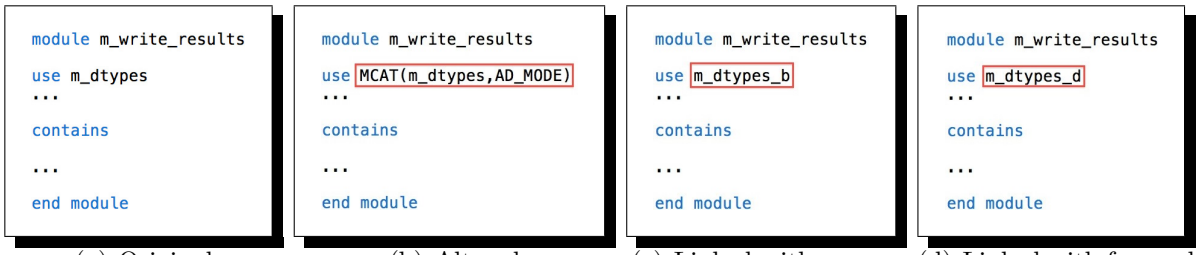
```
# FIRST DERIVATIVES
ifneq ($(AD_MODE),)
ad1:=$(shell \
  if [ $(AD_MODE) -eq 1 ] || [[ $(AD_MODE) -ge 11 && $(AD_MODE) -le 12 ]]; \
    then echo "d"; \
  elif [ $(AD_MODE) -eq 2 ] || [[ $(AD_MODE) -ge 21 && $(AD_MODE) -le 22 ]]; \
    then echo "b"; \
  fi)
endif
ifeq ($(ad1),d)
  # Tangent
  m1:=m03_adkeep%
  m2:=m06_bruteforce%
  m3:=m05_residual$(CP1_FS)%
  ...
  AD1_CMD:=$(m1)keep_grid_t_elem
  AD1_CMD+=$(m2)brute(x_vrt)\(cl cv)
  AD1_CMD+=$(m3)residual$(CP1_FS)(cv)\(res)
  ...
endif
ifeq ($(ad1),b)
  # Reverse (adjoint)
  m1:=m05_residual$(CP1_FS)%
  m2:=m05_residual$(CP2_FS)%
  m3:=m02_clcdcp%
  ...
  AD1_CMD:=$(m1)residual$(CP1_FS)(cv)\(cv res)
  AD1_CMD+=$(m2)residual$(CP2_FS)(vol nds_iedg nds_bvrt x_vrt x_fce x_bfce \
           cvin dv)\(vol nds_iedg nds_bvrt x_vrt x_fce x_bfce cvin dv res)
  AD1_CMD+=$(m3)lift_drag_3D(q bwt flowVec)\(cl cd)
  ...
endif
```

```
# AD TOOL
tapn_cmd=tapenade -head "$(1)" -$(2) \
  -diffvarname "$(3)" -difffuncname "$(4)" \
  -ext $(TAPN_LIB)PUSHPOPGeneralLib -extAD $(TAPN_LIB)PUSHPOPADLib \
  -inputlanguage fortran90 \
  $(addsuffix .f90, $(basename $^))
```

Figure 3.3: Rules [top] and call [bottom] to the AD tool.

3.4.3 Post–processing and linking

After applying AD and generating the desired sensitivity code, the latter needs to be linked with the the source code and compiled. The biggest challenge in this is the



(a) Original (b) Altered (c) Linked with reverse (d) Linked with forward

Figure 3.4: Example of modular program structure and alteration for correct linking with AD generated sensitivity code. From left to right: original module, altered module (which will be preprocessed) and correct linking with the reverse or forward module (via the equivalent *make* command).

automatic correction of included modules in the remaining source code. To make this more clear, suppose the module *m_write_results*, which includes subroutines for writing out results of the program in desired formats for post processing. This module is not differentiated but does include others that are, for example module *m_dtypes*, which describes the derived data types used in the program, Figure 3.4. After differentiating, the definition of the original (not differential) derived data type will be included both in the original and differentiated version of *m_dtypes*. Not both can be used though, since the compiler expects just one definition. To overcome this problem, a preprocessor can be used to parse inserted concatenation macros in the source code, Figure 3.4. These macros automatically change the modules used in every of the remaining non differentiated source code, according to the compilation command. With this logic implemented in the source code, any version of the sensitivity code can be compiled along with the rest of the program with no extra effort in changing the source code manually.

Another issue that can arise is the introduction of the non supported features into the preprocessed source code files. For example, the *GNU C* preprocessor was developed to handle the C, C++ and objective C programming languages. It can parse others as well, such as Fortran 90/95, but the output might be not suitable for compilation due to the introduction of C syntax comments. This problem can be solved by using search/replace commands, such as the stream editor *sed* [43], which removes these comments from the files.

A note here would have to concern the possibility to have various sensitivity codes at the same time. It has already been shown that the sensitivity code can be in either a forward or adjoint mode. It might be of interest to have both versions available in the software, e.g. for validation purposes. When using modules though, this would create a linking problem. The original module `mod` would be `mod_d` or `mod_b` in the cases of tangent and adjoint respectively and those would include original and differentiated information.

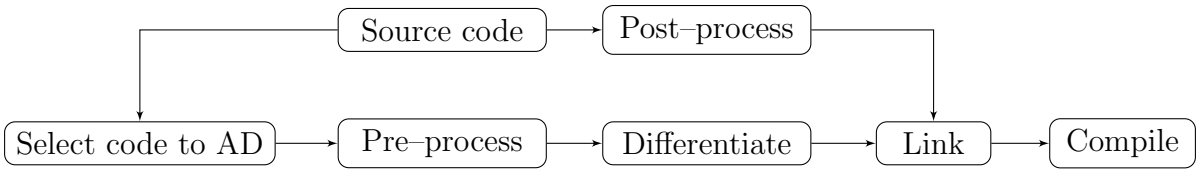


Figure 3.5: Automation algorithm for sensitivity code generation, linking and program compilation.

For example the derived data type `dtype` would be defined in both versions, preventing compilation. Instead of this, differentiation over differentiation can be used, a logic that will be discussed further in Chapter 5. If the first differentiation was in reverse and `mod_b` was the product, a second forward differentiation can be applied on that to acquire the tangent version of `dtype`. The final product of such an operation is the module `mod_bd`, which includes `dtype (original)`, `dtype_b` (adjoint) and `dtype_d` (tangent), allowing compilation this time.

Following all the steps described in the previous subsections, the generation of sensitivity code via AD can be completely automated, apart from the initial step of adjusting the source code to be AD parse–able. The entire process is summarised in the algorithm of Figure 3.5. As discussed, the *Makefile* can be used as the driver of all the operations, resulting in a compile–like generation of the sensitivity code.

3.5 Verification

Once the sensitivity code has been generated and integrated into the rest of the software, its results should be verified before actually being used in an optimisation algorithm. This could involve just a single function or the entire algorithm which computes the derivative of the cost function with respect to the design variables. The verification process can be performed in two steps by first verifying forward and reverse mode computed gradients versus finite differences and then by testing if reverse and forward mode match to machine precision [46]. Although Finite Differences (FD) can be accurate only to a limited extent of decimal digits, they can indicate the magnitude of gradients, which ought to match with those computed via AD generated sensitivity code. In the context of **mgOpt**, verification was carried out both at an individual function (when required, typically during debugging) and entire gradient computation level.

A single function verification can be demonstrated using the subroutine `calclift` of the software, which computes the lift of a given geometry via surface integration. This can be differentiated in forward and reverse mode, producing the subroutines `calclift_d` and `calclift_b` respectively. The argument lists of those two subroutines would be

Method	Gradient
Central Finite Difference	**-0.16376939**68009167
Tangent	**-0.16376939404141336**
Adjoint	**-0.16376939404141336**

Table 3.1: Finite difference vs tangent and adjoint for an individual function.

Method	Gradient
Central Finite Difference	**3.030588**9714641
Tangent	**3.0305888221515**
Adjoint	**3.0305888221516**

Table 3.2: Finite difference vs tangent and adjoint for the gradient of lift wrt the angle of attack.

(presenting only the inputs and outputs of most importance) :

$$\texttt{calclift\_d}(\alpha, \rightarrow \alpha_d, C_L, \leftarrow C_{L_d}) \quad \text{and} \quad \texttt{calclift\_b}(\alpha, \leftarrow \alpha_b, C_L, \rightarrow C_{L_b})$$

where "$\rightarrow$" and "$\leftarrow$" represent input and output respectively. Setting $\alpha_d = 1$ and $C_{L_b} = 1$ the two subroutines compute the derivatives $C_{L_d} = \frac{\partial C_L}{\partial \alpha}$ and $\alpha_b = \frac{\partial C_L}{\partial \alpha}^T$, which should match to machine precision [46]. The sensitivities via FD, tangent and adjoint are presented in Table 3.1, verifying the correct forward and reverse differentiation of `calclift`. The limited accurate FD provide qualitative information on the gradient magnitude, which matches in order with the AD ones.

The verification of most importance though is that of the entire algorithm that computes the gradient of the cost function with respect to the design variables. This was performed for every cost function of **mgOpt** and is going to be presented here for the lift function; the procedure is the similar for any other. For this, the NACA 0012 airfoil at Mach number 0.43 and angle of attack 2° is used. Table 3.2 shows the verified gradients between adjoint, tangent and FD. For FD, a perturbation δx stability study was carried out and the value of 10^{-5} was used. A more interesting thought is to exploit the properties of AD and the direct differentiation–adjoint equivalence (3.6). AD delivers machine accurate sensitivity code so gradients computed either in forward or reverse mode should match in every iteration of the time marching scheme, which drives tangent and adjoint residuals to zero. This relation was exploited using **mgOpt** and the results of Table 3.3 verify it.

Iteration	Forward mode	Reverse mode
1	1.5693931118856	1.5693931118856
2	2.5791035355540	2.5791035355540
3	3.2301257117945	3.2301257117945
4	3.6606833916720	3.6606833916720
5	3.9529170141718	3.9529170141718
6	4.1559630103599	4.1559630103599
7	4.2996992666559	4.2996992666559
8	4.4027343761762	4.4027343761762
...	...	...
10000	3.0305888221497	3.0305888221497
...	...	...

Table 3.3: Tangent vs adjoint gradient

3.6 Performance acceleration

The previous paragraphs presented the procedure of automated sensitivity code generation via AD (after an initial preparation) and the equivalent verification. The next challenge in using AD derived sensitivity codes and especially in reverse (adjoint) mode is to examine performance in terms of runtime and memory, which is the subject of this section. The next three subsections investigate this in terms of ways to use AD and linking of the automatically generated sensitivity code with typical acceleration mechanisms of the flow, such as multi–grid and the block Jacobi pre-conditioner.

3.6.1 Hand assembled adjoint code

As a first approach to accelerating the performance of AD generated adjoint code, the way that the AD tool is used and the assembly of the sensitivity code with the rest of the program are going to be considered. The simplest approach is to directly apply AD to a source code that has been prepared to be AD parse–able (Section 3.4.1) and link the source and sensitivity codes. Such an approach is referred to as **black box** application of AD and is illustrated in Figure 3.6. It can be applied on the AD–prepared source code without any further modification or directions to the AD tool. In the case of tools using *taping* [46, 47, 10, 20] (read and write to memory operations) such as *Tapenade* [101], this would generate sensitivity code with a long tape and therefore high memory and runtime demands. Alternatively, instructions to the tool can help reducing that length and resulting in a more efficient code. Those can be inserted in the source code

Figure 3.6: Black box use of an AD tool. No extra effort is required by the user on the source code, apart from its initial preparation so as to be AD parse–able.

```
!$AD II-LOOP
    do i=1,n_dx+1
      tmpv=cv(i,:)
      if(i>1)tmpv=tmpv/cv(1,:)
      gdtemp(:,:)=gd(:,i,:)
      call gradient( &
        n_dx,n_edg,n_vrt,n_bset,n_bvrt, &
        vrt_edg,bvrt_bset,vrt_bvrt, &
        nds_iedg,nds_bvrt, &
        gcr,vol,tmpv,gdtemp)
      gd(:,i,:)=gdtemp(:,:)
    end do
```

Figure 3.7: Insertion of pragmas to accelerate AD code performance. Here, the pragma `$AD II-LOOP` [101] informs *Tapenade* [101] of the following loop's iteration independence, leading to a sensitivity code with less stack operations and lower memory requirements.

in the form of directives and pragmas[1], which inform the tool about specific properties of the source code at certain points. For example, the pragma `$AD II-LOOP` [47] informs *Tapenade* that the loop to follow is iteration independent and the use of it in a part of **mgOpt** is presented in Figure 3.7. Such directives though can only be applied if the software developer has a very good understanding of the underlying operations and be confident of the correct positioning the instructions. Otherwise, the AD tool might be misled. Therefore, such a use of AD is no longer *black box* but rather **grey box**, since it requires effort from the user into altering the source code, Figure 3.8. This would still be though a rather brute–force application of AD (similar to black–box), since the user knows nothing about the internal assembly of the sensitivity code but accepts the result of the AD tool as it is.

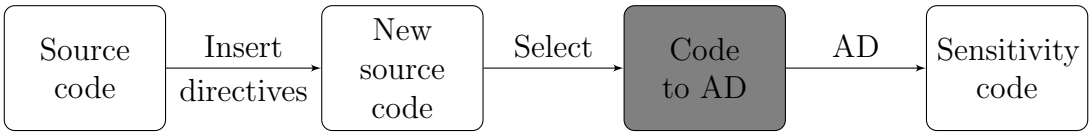


Figure 3.8: Grey box use of an AD tool. Extra effort is required by the user of the tool regarding insertion of directives and pragmas in the source code. The result is less stack operations and lower memory requirements [47].

An more efficient approach to using AD is to differentiate only the necessary parts

[1]Instructions on the available directives and pragmas and their use can be found in the AD tool's user manual.

of the source code and hand assemble the sensitivity algorithm [22, 38, 19]. Although AD provides a correct sensitivity computation algorithm, this will most probably not be of best possible performance, as the tool cannot identify all the points where the latter could be improved. By hand assembly, this problem is bypassed but again a good understanding of the underlying operations is required by the code developer. Although it can be time consuming to hand assemble the sensitivity algorithm, this is performed only once and thereafter the AD code generation, linking and compiling chain will be automatic. In the case of **mgOpt**, the most runtime expensive part of the code is the iterative solver that computes the flow. The equivalent algorithm is summarised in Figure 3.9, where the arguments of most importance are included. `Q` are the flow variables, `X` the coordinates of the volume grid nodes, `Nrm` the edge and boundary normals, `R` the residuals of the flow equations and `J` the cost function. The symbols $\rightarrow$ and $\leftarrow$ denote inputs and outputs respectively. This algorithm computes flow and cost function. It can be provided to an AD tool for brute–force differentiation, which would result in the sensitivity code of Figure 3.10. The over–line implies adjoint quantities and subroutines differentiated in reverse mode. In this way, the reverse differentiation traces the sensitivity backwards from the cost function through all iterations to the mesh coordinates. This form of the adjoint pseudo–time stepping loop contains the computation of the sensitivity of the residuals with respect to the perturbation of the normals ($\overline{\texttt{Nrm}}$), which is the term f^T in equation (3.11). This computation though only needs to be performed once, after the adjoint solution is converged and thus its repeated computation is not necessary and increases computational and runtime cost. Also, the adjoint variables $\overline{\texttt{Q}}$ will always be initialised from the reverse differentiated cost function (see $\overline{\texttt{cost\_function}}$), removing the possibility of initialisation with a pre–computed solution. Examining the algorithm of Figure 3.10 though, the following can be observed: i. the function $\overline{\texttt{cost\_function}}$ only adds the source term $g = \frac{\partial J}{\partial U}$ to the adjoint residuals and ii. the function $\overline{\texttt{update}}$ would do exactly the same operation to update the adjoint solution as `update`, as far as local time stepping is concerned. Therefore, the form of the primal time stepping algorithm could be reused, in the form of Figure 3.11, where only the subroutine `residual` has been differentiated once with respect to `R` and once with respect to `Nrm`, resulting in the functions $\overline{\texttt{residual\_r}}$ and $\overline{\texttt{residual\_nrm}}$ respectively. It must be noted that the positions of $\overline{\texttt{Q}}$ and $\overline{\texttt{R}}$ have been manually swapped in $\overline{\texttt{residual}_\texttt{r}}$ so as to maintain the logic of adjoint variables and residuals respectively. Also, the original `update` function is used, through which the adjoint variables are updated as :

$$\overline{Q}^{i+1} = \overline{Q}^{i} - \delta t \cdot \overline{R}^{i} \tag{3.12}$$

```
call initialise_flow ( ←Q )
call metrics ( →X, ←Nrm )
do nIter = 1,mIt
  call residual ( →Q, →Nrm, ←R )
  call update ( →R, ←Q )
end do
call cost_fun ( →Q, →Nrm, ←J )
```

Figure 3.9: Main structure of a compressible flow solver.

```
Q̄ = 0; J̄ = 1
call cost_fun ( →Q, ←Q̄, →Nrm, ←N̄rm̄, ←J, →J̄ )
do nIter = mIt,1,-1
  call update ( →R, ←R̄, ←Q, →Q̄ )
  call residual ( →Q, ←Q̄, →Nrm, ←N̄rm̄, ←R, →R̄ )
end do
call metrics ( →X, ←X̄, ←Nrm, →N̄rm̄ )
```

Figure 3.10: Brute–force differentiation of a compressible flow solver.

The values of δt used are those computed during the flow computation. The source term g must be added to the adjoint residuals before the solution update. In this way, the adjoint pseudo-time stepping loop follows the same calculation order as the primal and the same acceleration techniques can be used. Furthermore, the adjoint solution $\overline{\mathtt{Q}}$ can be initialised with any previous solution (hot–start) and not only through $\overline{\mathtt{update}}$. This will characterise such a set up very suitable for the one-shot approach in optimisation, as it will be explained later in Chapter 4. This form of the adjoint time–stepping shall be refereed to as "primal time-stepping" or PTS adjoint. Such an approach to using AD requires a large effort, knowledge and understanding from the software developer, who will now be using the AD tool only when required and not in a brute–force manner. This could be described as a **white box** use of AD, Figure 3.12. Again though, this preparation must be performed only once and then the entire chain of differentiating, linking and compiling will be fully automatic by using the methodology described in Section 3.4. Using the

```
call cost_fun ( →Q, ←g, →Nrm, ←N̄rm̄, ←J, 1 )
do nIter = 1,mIt
 call residual_r ( →Q, ←R̄, Nrm, ←R, →Q̄ )
 R̄ = R̄ + g
 call update ( →R̄, ←Q̄ )
end do
call residual_nrm ( →Q, ←Q̄, →Nrm, ←N̄rm̄, ←R )
call metrics ( →X, ←X̄, ←Nrm, →N̄rm̄ )
```

Figure 3.11: Hand assembly of the sensitivity algorithm.

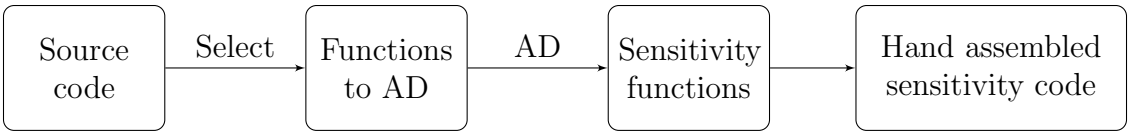


Figure 3.12: White box use of an AD tool. Maximum effort is required by the user which results though in higher performance sensitivity code from AD.

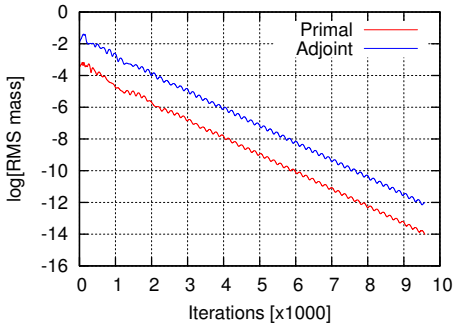


Figure 3.13: Demonstration of correct transposition of the Jacobian matrix **A**. Primal and adjoint converge at the same rate.

white–box approach (with inserted directives) the runtime improvement compared to the brute–force grey–box can be observed in Table 3.4. Such a performance is now more competitive with fully hand derived continuous [52, 86, 85] or discrete adjoint codes [38, 70]. In addition, this methodology is guaranteed to converge at the same rate of the primal for steady flows (AD produces the exact transpose of the Jacobian matrix **A**), which is might not be the case for hand derived adjoints due to human error or assumptions and simplifications. Figure 3.13 demonstrates this property on a NACA 0012 airfoil.

	Runtime (sec)	**Cost**
Primal	9.6886053	1.0000000
PTS adjoint	25.517594798	2.6337738
Brute force adjoint	40.826551312	4.2138729

Table 3.4: Runtime for the primal, PTS and brute–force grey–box adjoint pseudo-timestepping loop.

It shall be noted that, this approach also involves some extra effort in the differentiation preparation. As discussed above, functions might need to be differentiated more than once, when differentiating with respect to different arguments. An example of this is the function `residual`, which was differentiated once with respect to `Q` and once with respect to `Nrm`, Figure 3.11. If not otherwise instructed, the AD tool would in both cases produce routines with the same name, in this case `residual_b`. The same would happen

```
# AD FILES
ad_src:=$(notdir $(filter %_.f90, $(src)))
CP1_FS:=_r
CP2_FS:=_nrm
ifneq ($(CP1_FS),)
  CP_SRC:=$(subst _.,$(CP1_FS)_.,$(ad_src))
endif
ifneq ($(CP2_FS),)
  CP_SRC+=$(subst _.,$(CP2_FS)_.,$(ad_src))
endif
ad_src+=$(CP_SRC)
```

Figure 3.14: Duplication of the source code for multi-differentiation.

```
ifneq ($(CP1_FS),)
%$(CP1_FS)_.f90: %_.f90
   sed "s/^\([ ]*\(module\)[ ][^ ]*\)/\1$(CP1_FS)/g;s/^\([ ]*\(use\)[ ][^ ]*\)/\1$(CP1_FS)/g;s/^\ \
        ([ ]*\(subroutine\)[ ][^ ]*\)/\1$(CP1_FS)/g;s/^\([ ]*\(call\)[ ][^ ]*\)/\1$(CP1_FS)/ \
        g" $< > $@
endif


ifneq ($(CP2_FS),)
%$(CP2_FS)_.f90: %_.f90
   sed "s/^\([ ]*\(module\)[ ][^ ]*\)/\1$(CP2_FS)/g;s/^\([ ]*\(use\)[ ][^ ]*\)/\1$(CP2_FS)/g;s/^\ \
        ([ ]*\(subroutine\)[ ][^ ]*\)/\1$(CP2_FS)/g;s/^\([ ]*\(call\)[ ][^ ]*\)/\1$(CP2_FS)/g" \
        $< > $@
endif
```

Figure 3.15: Pre–processing of the duplicated source code.

for all functions called and the containing module and this would constitute a compiler rules violation. Most of the AD tools provide functionalities, which change the name of the differential functions but this would not be enough, as this would be applied only on the main function and its module. All the internally called functions would still have the same name, creating a compilation problem. In order to bypass this problem, duplicating and preprocessing of such parts of the source code is proposed. This is performed completely automatically, once the equivalent commands have been implemented in the *Makefile*. These commands include operating system shell directions for multi-copying of the source code with the desired suffixes, instructions to the pre-processor, which will change all the names of functions and modules called accordingly as well as *Makefile* scripting for the addition of the duplicated code to the definition of the source code to be provided to the AD tool. A sample of such a parts of the *Makefile* is presented in Figures 3.14 and 3.15. For the case of `residual`, these commands would generate the two duplicates needed: `resibual_r` and `residual_nrm`. Last but not least, although these duplicated modules do not actually exist in the source code, the latter needs to include them and therefore the equivalent statements need to be added to the differentiated code. Such an example is presented in Figure 3.16.

```
module m04_design
   use MCAT(m05_residual_n,AD_FS)
   use MCAT(m05_residual_n,AD_FS)
   use MCAT(m02_clcdcp_u,AD_FS)
   use MCAT(m02_clcdcp_u,AD_FS)
   ...
end module
```

Figure 3.16: Use of copied modules in non-differentiated source code. *AD_FS* is defined at compile time, determining suffix/type of the relative module's differentiation (tangent or adjoint) and *MCAT* directs the C-preprocessor to concatenate the module name with the differentiation suffix.

3.6.2 Primal multi–grid

The convergence of the adjoint equations (3.5) can also be accelerated by using the multi–grid methodology, Section 2.8.1. In the context of the developed software **mgOpt**, the author examined the use of the primal (not differentiated) geometric multi–grid for the adjoint. This was formed in the same logic as for the flow, but using the adjoint variables and residuals. Following the nomenclature of Section 2.8.1 and the FAS methodology [15], the multi–grid operations for the adjoint follow the next steps :

1. *Smooth adjoint errors on the finer level.*

$$v^n = v^n + \Lambda \left[(g^T)^h - (\mathcal{A}^T)^h v^n \right] \tag{3.13}$$

2. *Add the adjoint source term to the finer level adjoint residuals.*

$$\bar{R}^h = \bar{R}^h + g^T \tag{3.14}$$

3. *Transfer the adjoint solution and residuals to the coarser level.*

$$v^H = I_h^{H,U} v^h \;\text{ and }\; \bar{R}^H = I_h^{H,\bar{R}} \bar{R}^h \tag{3.15}$$

4. *Prolong the coarser level adjoint correction to the finer level.*
   After smoothing on the coarsest level :

$$v^H = v^H + \Lambda \left[(g^T)^H - (\mathcal{A}^T)^H v^H \right] \tag{3.16}$$

   the coarser level correction is prolonged to the finer level :

$$v^h = v^h + I_H^h (v^H - I_h^{H,U} v^h) \tag{3.17}$$

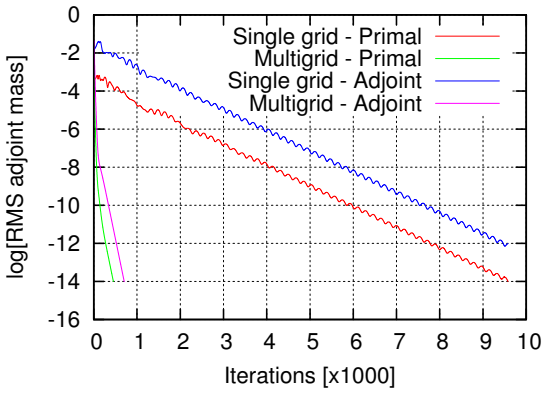


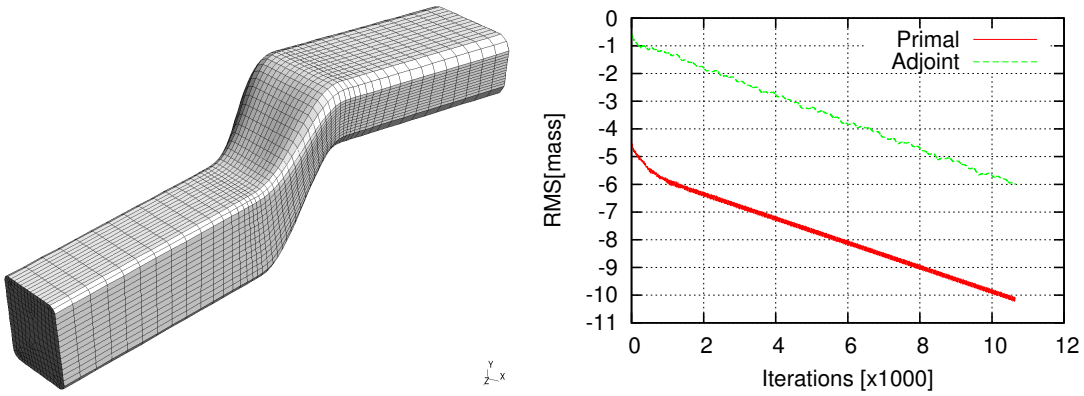
Figure 3.17: Use of geometric multi–grid on the adjoint. Primal and adjoint are accelerated at the same ratio.

In the equations above, v and $\bar{R}$ are the adjoint variables and residuals respectively. This methodology has been implemented in **mgOpt**, resulting in converse acceleration of the same order of magnitude as for the flow. An example is given in Figure 3.17.

3.6.3 Pre–conditioning

Section 2.8.2 discussed the use of the block Jacobi pre–conditioner [3] in the context of accelerating the convergence of the system of flow equations. In the same logic, the pre–conditioner could be used to accelerate the converge of the system of adjoint equations. The most accurate approach to this would be to differentiate the block Jacobi pre–conditioner in reverse and compute it in every step of the iterative solver based of the adjoint variables. Since though the transposed Jacobian matrix $\mathbf{A}^T$ of the adjoint system of equations (3.5) is the exact transpose of the Jacobian matrix $\mathbf{A}$ of the system of flow equations and therefore they share the same eigenvalues, it would interesting to examine if the adjoint system can be preconditioned by a block Jacobi matrix based on the primal. Using Newton's method, an intermediate adjoint solution v_i^n is updated as :

$$\begin{aligned} v_i^{n+1} &= v_i^n - \mathbf{A}^T \bar{R}_i^n \\ \Rightarrow v_i^{n+1} &= v_i^n - \left[\frac{\partial R_i}{\partial Q_i}(Q^n)\right]^{-T} \bar{R}_i^n \end{aligned} \tag{3.18}$$



(a) Discretisation (b) Convergence

Figure 3.18: Pre–conditioning of the adjoint with the block Jacobi matrix. Primal and adjoint converge at the same rate.

where $\bar{R}_i^n$ the residuals of the adjoint system. The pre–conditioning matrix in the case of the adjoint system would therefore have the form of :

$$\bar{\mathcal{B}}_i = \Delta t_i \begin{bmatrix} \frac{\partial R_1}{\partial Q_1}(Q^n) & 0 & 0 & 0 & 0 \\ 0 & \frac{\partial R_2}{\partial Q_2}(Q^n) & 0 & 0 & 0 \\ 0 & 0 & \frac{\partial R_3}{\partial Q_3}(Q^n) & 0 & 0 \\ 0 & 0 & 0 & \frac{\partial R_4}{\partial Q_4}(Q^n) & 0 \\ 0 & 0 & 0 & 0 & \frac{\partial R_5}{\partial Q_5}(Q^n) \end{bmatrix}^{-T} \overset{(2.90)}{=} \mathcal{B}_i^T \tag{3.19}$$

So, the adjoint solution is updated by:

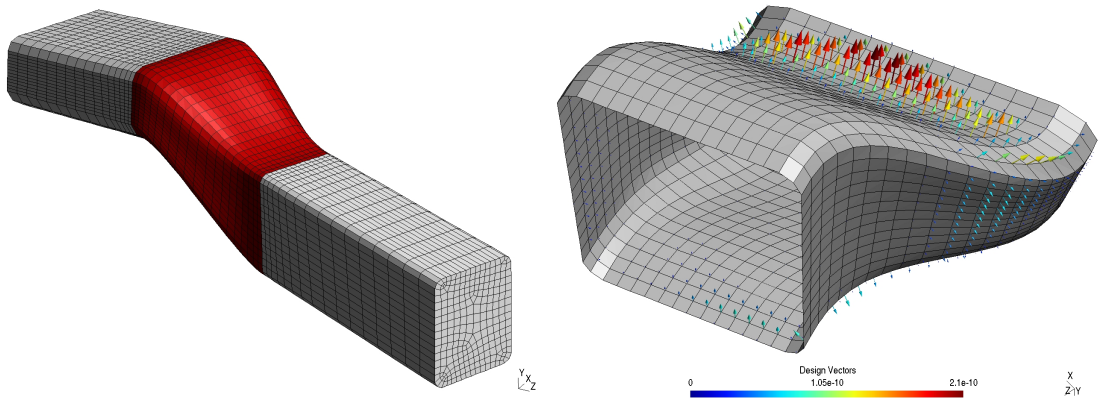
$$v_i^{n+1} = v_i^n - \bar{\mathcal{B}}_i \bar{R}_i^n \tag{3.20}$$

$$\overset{or}{\Rightarrow} v_i^{n+1} = v_i^n - \mathcal{B}_i^T \bar{R}_i^n \tag{3.21}$$

This algorithm has been implemented in **mgOpt** by the author and such a pre–conditioner for the adjoint system results in a similar convergence rate between primal and adjoint. This is demonstrated in Figure 3.18 for a laminar flow case through an S-Bend duct, which was discretised with 16000 hexahedral elements.

3.7 Examples of sensitivity vectors

Once an adjoint solver has been implemented and verified, it computes the sensitivities of a cost function J with respect to any number of design variables α. These gradients provide information on how the shape should change so that the cost function is driven to



(a) Design space (red) (b) Sensitivity vectors

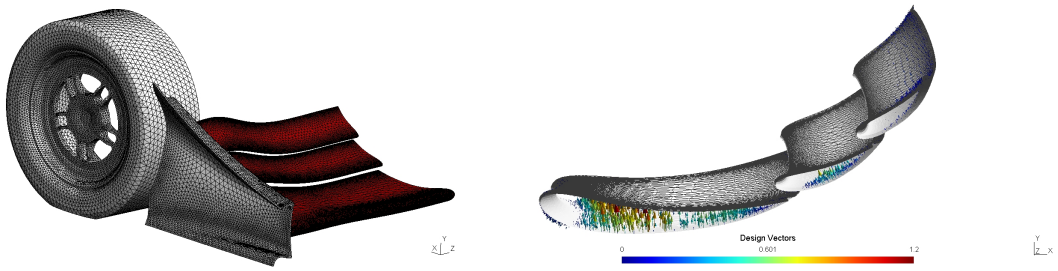
Figure 3.19: Sensitivity vectors on an S-Bend duct. Courtesy of VolksWagen.

an extremum, e.g. a minimum, and can either be used as guidance to manually change the shape or to guide an optimisation algorithm (Chapter 4). In this section, various examples of sensitivity vector maps are presented by using the automatically generated AD adjoint of **mgOpt** and the cost functions available (see Section 2.7).

A first example is given using the low speed, ducted laminar flow case through the S-Bend of Section 2.9.3. This geometry was provided by Volkswagen and, for this examples, only the bended fraction was defined as the design space. It should be mentioned though that even if the entire solid wall was the design space, the cost of computing the sensitivities using the adjoint methodology would be the same, as explained in Section 3.3. The design vectors of Figure 3.19 indicate how the surface should be moved so that the pressure drop through the entire duct is minimised. Most of these vectors point outwards, which is physically valid as, increasing the volume of the bended section would lower speed and increase pressure. Some of these indications could possibly be predicted by an experienced CFD analyst, who however would not be in a position to predict the best possible shape. Before moving to the next example it is worthwhile mentioning that the design surface in this case does not participate in the definition of the cost function but only contributes to the change of the flow.

Another example can be given for the external flow case over the front wings of the student formula car, in which the wheel is also added. The cost function was the down force coefficient constrained by the drag coefficient. The target is to maximise the down force at the speed of 100 km/h, where the maximum acceleration appears, in order to increase traction but without increasing drag. The cost function for this case was defined as :

$$J = \frac{\epsilon_1}{c_{DF}} + \epsilon_2 \cdot c_D \tag{3.22}$$



(a) Design space (red) (b) Design vectors

Figure 3.20: Design vectors on the front wings of a student formula car.

where c_{DF} and c_D the down force and drag coefficients respectively. The parameters ϵ_1 and ϵ_2 are weighting constants (in this case $\epsilon_1 = 1$ and $\epsilon_2 = 10$). For this example, the design space is defined by the wings.

Sensitivity vectors like the above provide valuable information that lead to a better design. They can be coupled with an optimisation algorithm (Chapter 4) and lead to the optimum shape automatically. Alternatively, this information may also be used just as an indicator of manual changes because the use of an design algorithm is of lower interest. In the automotive industry for example, small improvements are of interest and the actual optimal shape is of secondary importance. Such improvements can be achieved by examining the sensitivity vectors and applying changes by hand. With the proposed methodology (Section 3.6) this would cost only almost two flow evaluations and is therefore very affordable. This logic can be demonstrated using Figure 3.21, which presents the design vectors on the VolksWagen Passat, as computed by **mgOpt** using the drag force as the cost function. Apart from the expected indications for smaller gaps between hood/wide–screen, windows/car body and handles as well as a smaller mirror, they also advise that a spoiler (or a lifted geometry) at the rear can reduce drag. If such indications are taken into consideration by the industry during the design and production process, more efficient products can be delivered to the public, with lower emissions and therefore a smaller impact on the environment.

3.8 Summary

The present chapter focused in presenting the adjoint methodology and its role in shape optimisation in CFD. After a brief description of the shape optimisation problem in CFD, the theories behind direct differentiation and the adjoint were presented. In this analysis, the property of the adjoint to be able to compute derivative information independently of the number of design variable proved that the adjoint is a very promising approach

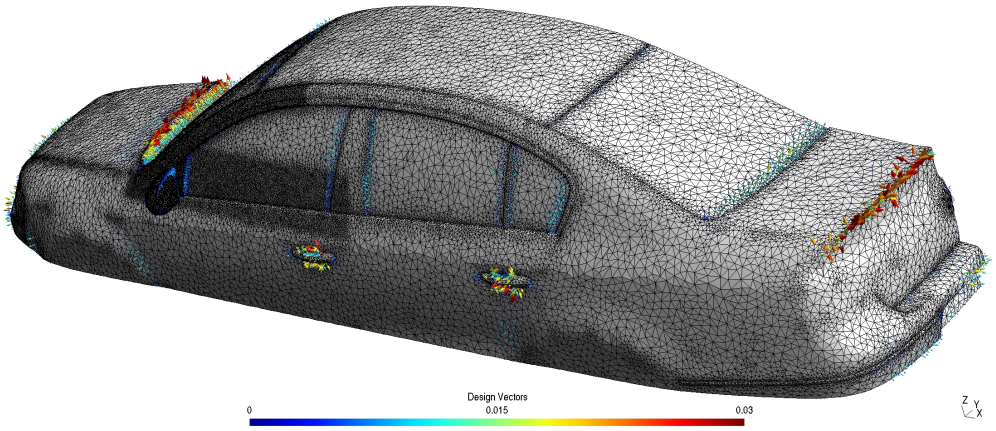


Figure 3.21: Design vectors on the Volkswagen Passat.

for large scale industrial shape optimisation cases.

The main interest in this chapter was to examine the automatic generation of adjoint sensitivity code via Automatic Differentiation (AD), using the high level modular language of Fortran 90/95. Until today, only relatively low level languages like Fortran 77 or low level Fortran 90 have been tested by the CFD community for source code transformation AD. The problems in this were described and way to overcome them were presented. The proposed state of the art methodology led to a versatile and full automated generation of adjoint code via AD for the developed software **mgOpt**, which makes use of modules, derived data types, pointers and all features of a modern object oriented language. All the above are controlled via a single *Makefile* and therefore convenient sensitivity code maintenance is promoted.

After the generation of adjoint sensitivity code via AD, the computed gradients were verified, proving that the differentiation and the use of the generated code is correct. Issues of maintaining both adjoint and tangent linear sensitivity codes in the same software were outlined and a methodology to overcome the related problems was described.

Once all the above were discussed, the next bottleneck in using AD derived adjoint, the performance, was examined. It was presented that a hand assembled sensitivity code can improve performance a lot and that a brute–force use of AD shall be of second choice, if the knowledge and time to perform the by hand implementation is available. Apart from this, the use of multi–grid and pre-conditioners on the adjoint was demonstrated, methodologies that accelerate performance even further and can be coupled with a hand–assembled sensitivity code.

Last, examples of sensitivity vector were presented on various cases. These vectors indicate the shape change towards a better geometry. It was discussed how these can be used in order to manually change the shape but in the next chapter they will be coupled with an optimisation algorithm, in order to produce optimum shapes automatically.

Chapter 4

Optimization

4.1 Introduction

The previous chapters presented the relative work of the present research in computing fluid flow and sensitivities of a cost function J with respect to the design variables $\vec{\alpha}$. These sensitivities are computed via the adjoint methodology, which is characterised by its independence from the number of design variables. Automatic differentiation was used for the derivation of the adjoint system, which was hand assembled for performance acceleration. Once the sensitivities are computed, they can be used as indicator that suggests shape changes that will improve performance, or they can be coupled with an optimisation algorithm in order to automatically drive the shape to its optimum. This is the subject of the present chapter, which discusses the rest of the basic ingredients for such an algorithm.

4.2 Parametrisation

The first step towards defining an optimisation case is the parametrisation of the design space. In aerodynamic shape optimisation problems, this could be performed by using Bezier lines [35], Splines [2, 31], B-Splines [28, 59, 60], NURBS (Non-uniform rational basis splines) [89] e.t.c. .These are smooth polynomial functions, which preserve the smoothness of the surface and are modified by their control points. An example of such a parametrisation can be observed in Figure 4.1, where a sphere is parametrised with NURBS. In such a parametrisation, the design variables of the optimisation problem are the control points of the parametric surface. The advantage of this approach is that the design surface always remains smooth, even if big design perturbations occur. Therefore, there is no need for the implementation of a smoothing algorithm to prevent

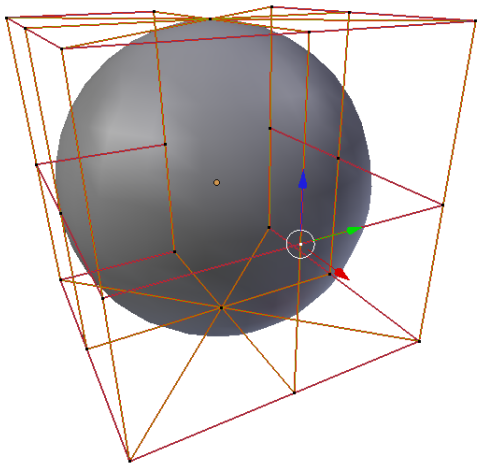


Figure 4.1: Sphere parametrised with NURBS. The black dots are the control points of the surface. Moving the latter, the form of the surface will change. Geometry created in *Blender* [13].

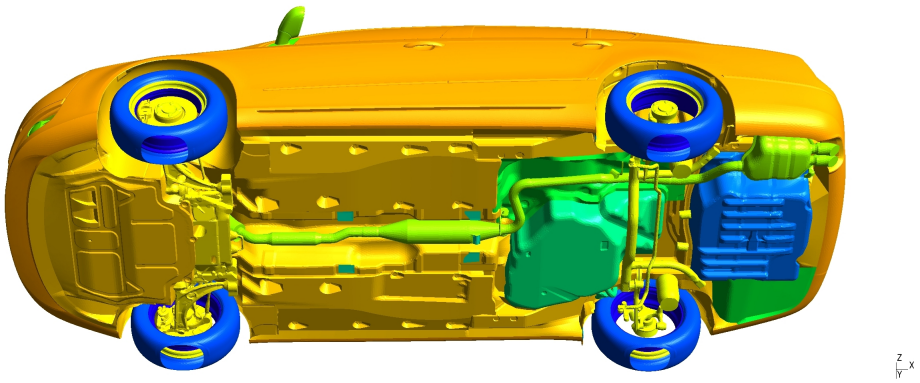


Figure 4.2: Complex geometry under a passenger car (source: Volks Wagen) . The parametrisation only with NURBS and the imposing of constrains is appears difficult.

the generation of noisy designs, during optimisation. On the other hand, it is very difficult and maybe impossible to parametrise complicated geometries with the approaches above. For example, the parametrisation of an under vehicle geometry, Figure 4.2, only with NURBS would be very difficult and take months of implementation. Apart from this, there are further issues with imposing constraints on the design space, which presents many challenges for complex geometries. Last but not least, a large number of control points would have to be introduced in order not to restrain the design space further, action which would introduce more complexity to the previous comments.

Instead of the approach above, in this thesis a ***node based*** parametrisation is used, where every node of the surface mesh of the design surface is a design variable, Figure 4.3. In the case of the present research, such nodes are allowed to move in the normal direction to the surface for reasons of simplicity. Otherwise, every variable can have from

one to three degrees of freedom ([x,y,z] directions) in the Cartesian coordinate system. The advantage of this approach is that, even the most complicated geometries, can be straightforwardly parametrised for optimisation. Apart from this, imposing constrains can be achieved faster than in the parametric surface description, at least in the form of *box* constrains (minimum and maximum values for the design variables). The main disadvantage of the node based parametrisation though is that, there is need for implementation of a surface smoothing algorithm, so as to prevent noisy surfaces, which are not manufacturable. The complexity of this can increase, when feature lines and specific characteristics of the design surfaces have to be maintained, e.g. lines which are fixed by the artistic designer of a car.

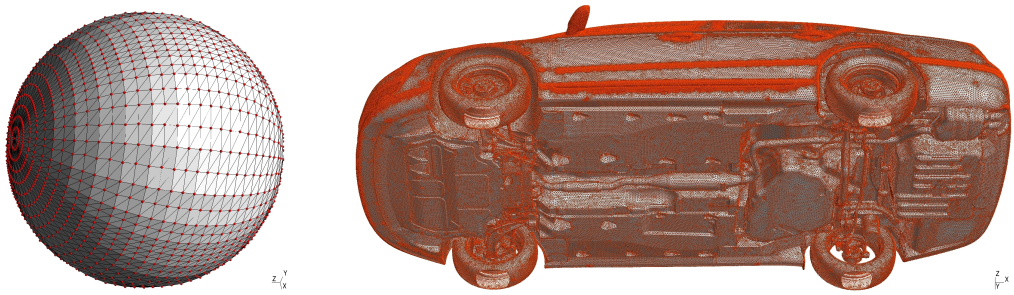
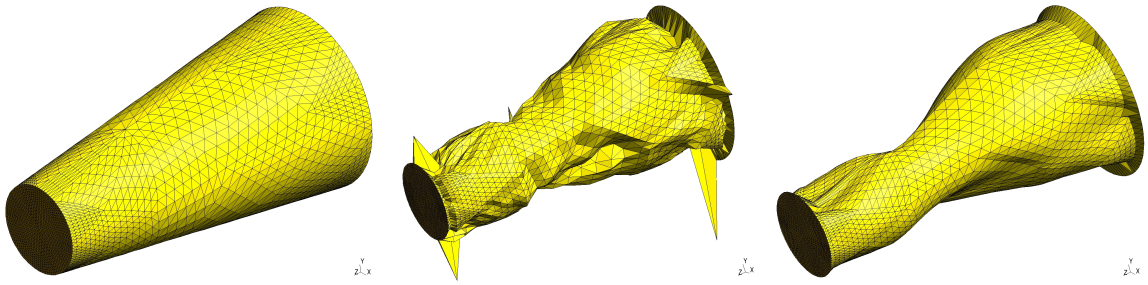


Figure 4.3: Examples of node based parametrisation on the sphere of Figure 4.1 [left] and the complex geometry of Figure 4.2.

After a parametrisation has been selected, the gradients of the cost function with respect to the design variables can computed using the adjoint methodology as discussed in Chapter 3. Those provide the direction towards which the design should move. The next step before perturbing the design surface is the computation of the design step (values of the design variables), which will determine the surface movement. Various algorithms have been proposed for this, with the most popular being *Steepest Descent* [17, 98], *Newton's method* [32, 14], *Quasi Newton's method* [16, 29] and *Conjugate gradients method* [49]. These algorithms are available today on the web, in the form of open source libraries. Proprietary libraries may also be found. The one used in this research is the Quasi Newton method in the form of the open source library L-BFGS-B [111] (`http://users.eecs.northwestern.edu/~nocedal/lbfgsb.html`), which was designed for limited memory systems and also provides a simple form of box constrains.

4.3 Smoothing

As discussed in Section 4.2, the surface mesh nodes of the design space are used as design variables in this thesis and not the control points of a parametrisation curve, such as



(a) Initial (b) Without smoothing (c) With smoothing

Figure 4.4: Effect of smoothing on the design surface. Initial shape [top left], perturbed surface **without** smoothing [top right] and **with** smoothing [bottom].

Splines, NURBS, etc. This parametrisation enables the representation of practically any potentially complex geometry and provides the opportunity to impose straightforward constrains on the design, at least in the form of box constraints. To raise the regularity of the computations though, smoothing has to be introduced for various aspects of the optimisation algorithm. These shall be discussed in this section.

The first element to be smoothed is the design surface. In a node based parametrisation, the perturbation of any design variable varies from those of the neighbouring nodes. This can result in noisy surfaces after the design perturbation, as in the example of Figure 4.4, which can lead to divergence or non manufacturable shapes. Introducing smoothing, this problem can be restrained and, when used along with scaling (Section 4.4), almost eliminated.

Another aspect of the design process that needs smoothing is the volume mesh. The perturbation of the design surface introduces deformation of the volume mesh and can decrease the mesh quality. This though would be undesirable as it can lead to computational inaccuracy, especially when boundary layers for viscous flows are used (such meshes are not going to be examined in the present thesis but this problem is similar for any type of mesh). This problem occurs for any type of parametrisation and not only in a node based context. Therefore, smoothing has to be applied on the volume mesh, in order to maintain the accuracy level of the computations. An example of such an operation can be observed in Figures 4.5 and 4.6. As it can be observed in Figure 4.5, where no volumetric smoothing is applied, the perturbation of the design surface distorts the volume elements associated with it to a big extent. This would introduce inaccuracies, especially in the case of viscous flows. Applying smoothing though can reduce this phenomenon. Figure 4.6 compares the same internal volume mesh with and without smoothing and, as it can be observed, the distortion effect is less significant when smoothing is used. In this example, only one smoothing iteration was used, which was

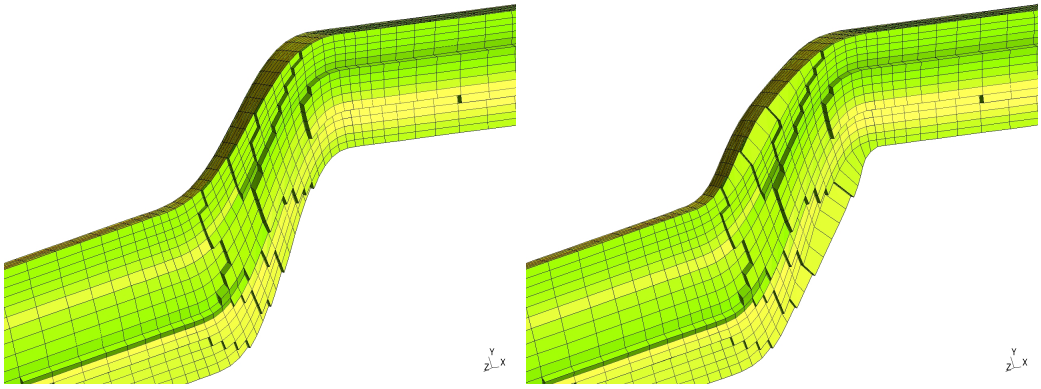


Figure 4.5: Initial [left] and not smoothed [right] volume mesh. The figures are plane cuts through an S-Bend.

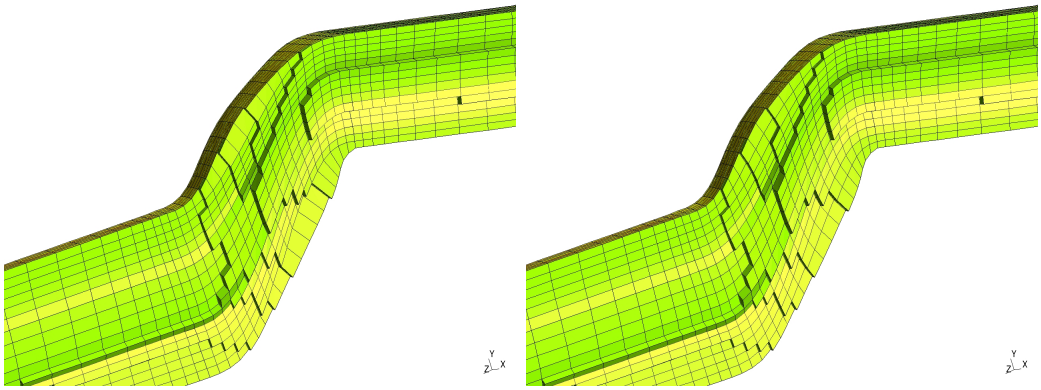


Figure 4.6: Effect of smoothing on the volume mesh. Not smoothed [left] and smoothed [right] volume mesh. The original mesh is presented in Figure 4.5.

able to demonstrate the smoothing effect. Should more smoothing iterations be used, the effect of distortion would be even smaller.

Last but not least, smoothing can also be applied on the gradients, in order to prevent extreme abnormalities in the deformation [97, 55] (e.g. Figure 4.4).

The next two subsections present the smoothing methodologies implemented and used throughout this thesis.

4.3.1 Implicit Sobolev smoothing

A first way of smoothing is the implicit *Sobolev* smoothing. This type of smoothing has been used in [54] by Jameson and Vasseberg to smoothen the gradients. It is applied through equation :

$$\psi_i^{j+1} - \frac{\beta}{2}\left(\psi_{i-1}^{j+1} - 2\psi_i^{j+1} + \psi_{i+1}^{j+1}\right) = \psi_i^j \tag{4.1}$$

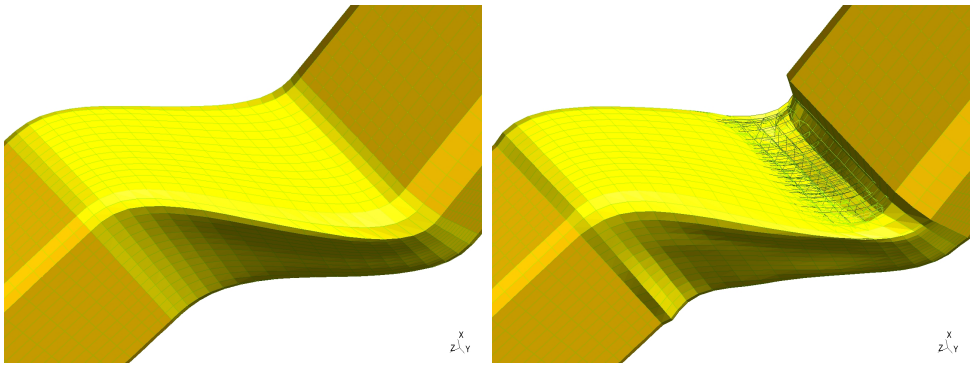


Figure 4.7: Problem of unscaled design. If the perturbation is large, smoothing cannot even it out in a logical number of smoothing iterations. Initial [left] and perturbed [right] geometry, where the design surface penetrates the volume mesh.

where ψ the quantity that needs to be smoothed, i the variable under examination and j the smoothing iteration number. The constant variable $\beta \in [0, 1]$ is an under relaxation factor. The disadvantage of this type of smoothing is that it affects a broad range of frequencies, that can compromise the accuracy of the design [55, 56]. An alternative formulation is given in the following paragraph.

4.3.2 Explicit Jacobi smoothing

Another smoothing methodology is the explicit *point-Jacobi* smoothing. It can be applied using equation (4.2).

$$\psi_i^{j+1} = \psi_i^j + \frac{\beta}{2}\left(\psi_{i-1}^j - 2\psi_i^j + \psi_{i+1}^j\right) \tag{4.2}$$

In comparison to implicit smoothing, explicit smoothing affects mostly the higher frequencies and has very little effect on the lower frequency modes. Apart from that, it is very straightforward to implement.

4.4 Design scaling

Section 4.3 revealed a problem that can occur after the movement of the design surface. The volume mesh gets distorted and its quality is reduced, Figure 4.5. One way to prevent this, is the use of smoothing on the volume mesh, Figure 4.6. If the perturbation of the design is rather large though, smoothing cannot even it out in a logical number of smoothing iterations. In such cases the quality of the mesh will remain low (after the perturbation) or even worse, problems of not matching shape and volume mesh can occur, like the one in Figure 4.7. In order to prevent this problem, scaling can be applied on the design variables, which will restrict the perturbations at a local level for each

design variable, equation (4.3) and Figure 4.8.

$$\alpha_s^i = \sigma \cdot^i \alpha^i \tag{4.3}$$

In equation (4.3), α^i are the unscaled design variable ($i = [1, ..., m]$, where m the number of design variables, σ^i the scaling factor for α^i and α_s^i the equivalent scaled design variable. The application of scaling may result in slower design convergence but it restricts

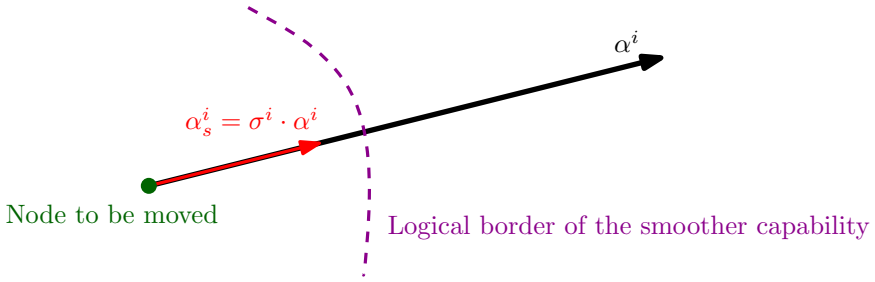


Figure 4.8: Logic behind the scaling of design variables.

abnormalities, like the one in Figure 4.7. In the context of this thesis, geometrical scaling is used, based on edge lengths. Equation (4.4) describes the computation of the scaling factors via this approach. Similar logic can be used based on other geometrical variables, such as boundary normals or volumes.

$$\sigma^i = \xi \frac{\sum_{j=1}^{N} L_j}{N} \tag{4.4}$$

N is the number of edges connected to the node/design variable (node based logic), L_j the length of edge j and $\xi \in [0, 1]$ a global scaling factor, which may or may not be used for further scaling (user defined parameter).

4.5 One shot methodology

An optimisation problem in CFD using the adjoint methodology would be described by the *Karush-Kuhn-Tucker* (KKT) system [57, 58] :

$$\mathbf{R}(Q, \alpha) = 0, \tag{4.5}$$

$$\mathbf{A}^T v = g, \tag{4.6}$$

$$\frac{\partial J}{\partial \alpha} + v^T f = 0, \tag{4.7}$$

This includes the solution of the primal (flow) (4.5) and dual (adjoint) (4.6) as well as the gradient equation (design) (4.7). When the KKT system is satisfied, the gradients are zero and the optimisation has reached an extremal point. An approach to converging this system to its solution is to converge the primal and dual to a large extent, before the gradients are computed. These gradients are of high accuracy, since they are based on accurate flow and adjoint. This methodology is accompanied by a relative runtime cost, which involves the computation of high fidelity gradients in each design step. This cost can be reduced by adopting the logic that the direction to which the gradients are pointing is more important that their magnitude. This means that less accurate gradients can be computed in each design step which will point to the direction of the extremal location. The closer to the minimum or maximum though, the more accurate the gradients need to do be. Following this methodology, flow, adjoint and design are converged simultaneously. This is called the ***one–shot*** methodology [55, 48] for gradient based optimisation. In order to use the one–shot methodology, an appropriate stopping criterion for the convergence of primal and dual has to be adopted. A simple approach is to use a specified number of iterations for primal and dual [48]. A more efficient approach was proposed by Jaworski et. al. [55], where the convergence of the primal and the dual is dynamically controlled via the norm of the gradients with the magnitude of the gradient, equation 4.8.

$$g_{RMS} = \sum \frac{|\nabla J|}{C} \tag{4.8}$$

where g_{RMS} the stopping criterion, J the cost function and C a user defined constant. Such a formulation automatically increases the accuracy of primal, dual and gradients with the number of design iterations. For this reason, this methodology is adopted in this thesis and to be precise, primal and dual are converged to a large extent at the first design iteration and, from there on, one shot is used, as described in the algorithm of Figure 4.9. An example of the acceleration that can be achieved is presented in Figure 4.10, for which the case of upcoming Section 4.6.1 was used .

4.6 Optimization results

In optimization cases, the optimal shape is usually not known and therefore there is no comparison for the output of the optimisation code. The only indication that the optimisation was successful is the performance improvement. Before such case can be examined though, the optimisation algorithm has to be tested and validated. For this reason, a few reverse engineering optimisation cases were constructed, where the final shape is known. Shall the optimisation algorithm end up in the known target shape,

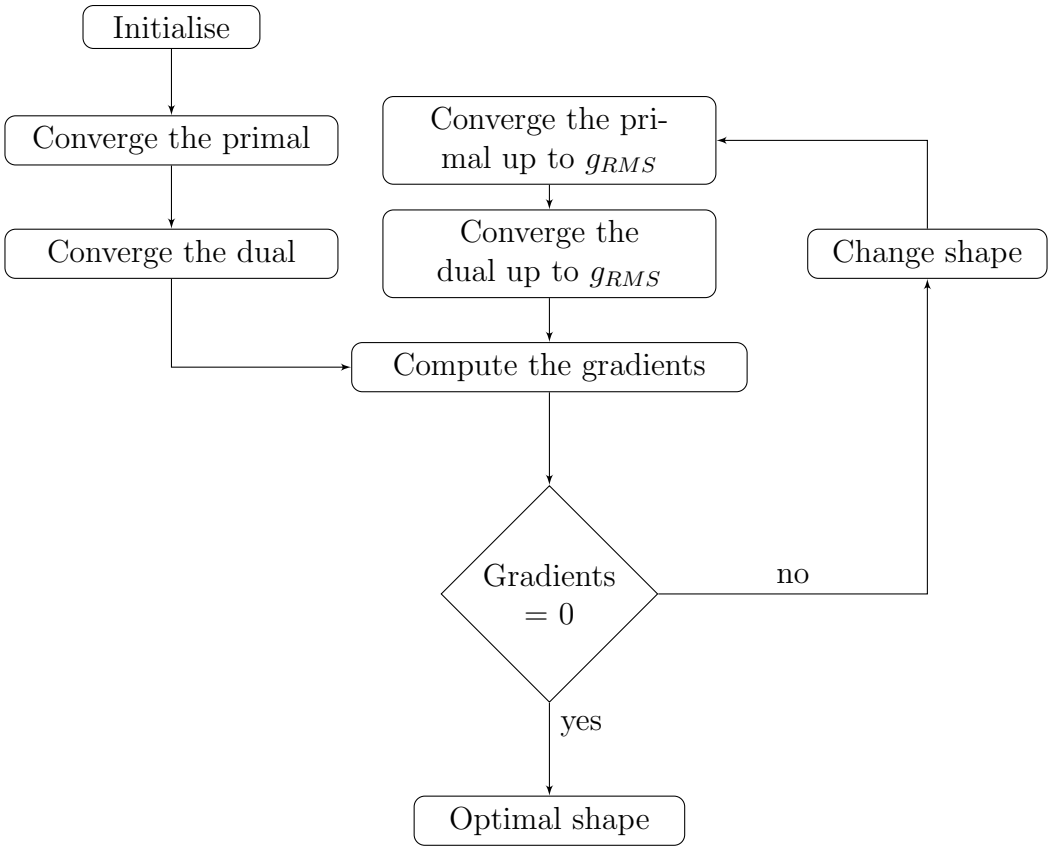


Figure 4.9: Dynamic one–shot approach used in **mgOpt** for converging the KKT system.

after beginning from another one, then it can be trusted. Three such cases are presented in the following subsections.

4.6.1 Pressure distribution recovery on a RAE 2822 airfoil

A first case for validation of the optimization algorithm is to perturb a node of a known airfoil, creating a small bump, and try to match this modified shape to the original one. In order to demonstrate this example, the airfoil RAE 2822 will be used. Both for the original RAE shape and the perturbed shape, set of four unstructured meshes of triangular elements is used for the multi–grid process, with 81 nodes on the surface, 4527 cells and 2310 nodes on the finest grid. As far as the one-shot convergence criterion is concerned (equation 4.8), the value of the constant C is set to 10^5, as proposed in [55]. The conditions are set to $Ma = 0.43$ and angle of attack $\alpha = 0.0°$. No smoothing is used in this case. This is a rather simple case, but it consists a solid optimisation validation case, one step before using the entire node based parametrisation, and is also of low cost. These are the main reasons why this is presented here. The original shape of RAE 2822 and its pressure distribution are presented in the Figure 4.11.

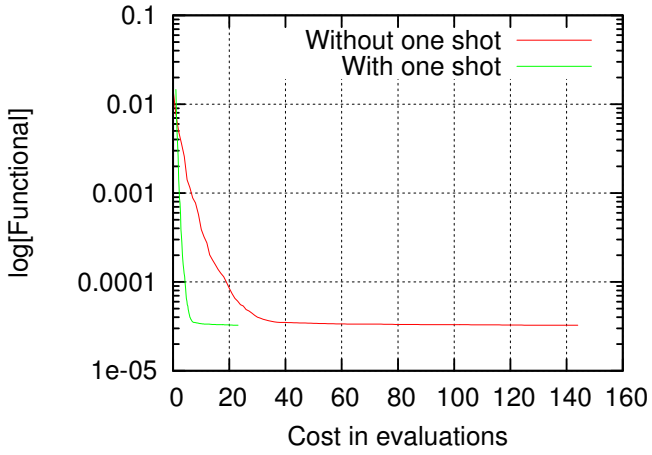


Figure 4.10: One–shot acceleration example

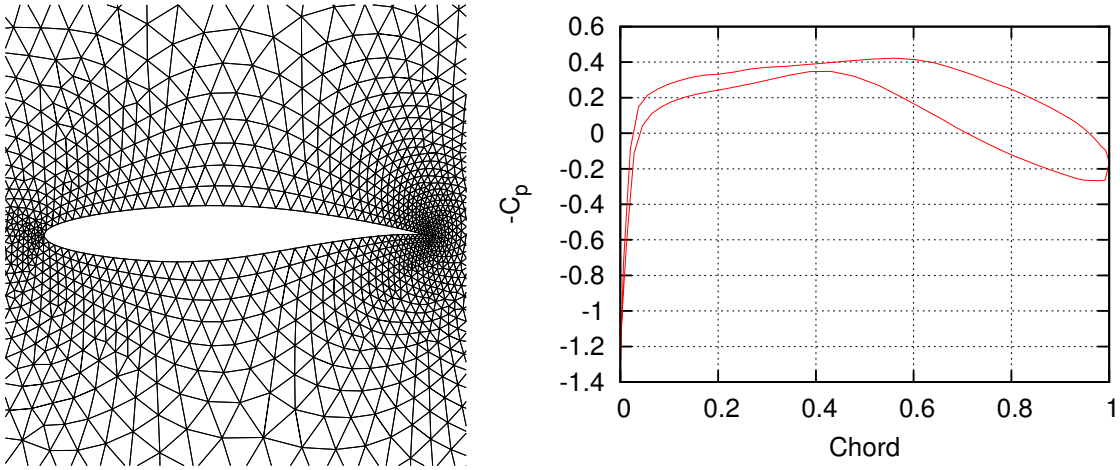


Figure 4.11: Original RAE 2822 shape and equivalent pressure distribution.

Perturbing a node on the front bottom part of the airfoil, one can get the perturbed shape and its initial pressure distribution, Figure 4.12.

The area, where the initial shape is perturbed, can be observed in more detail in the zoomed Figure 4.13. The node has been perturbed in the y-direction.

Running the optimization algorithm by using the L^2_{norm} of the total pressure, as described in Section 2.7.3, the perturbed node indeed returns to its original position and recovers the pressure distribution. The convergence of the functional and its gradient are presented in the graphs of Figure 4.14.

Using this simple test case, one can have a first indication that the optimization algorithm is functioning properly, before moving on to a more complicated test case (4.6.2) or a real optimization case, like the ones examines in later sections. The advantage of this test case is that the scale of the problem is small and so the convergence of the shape

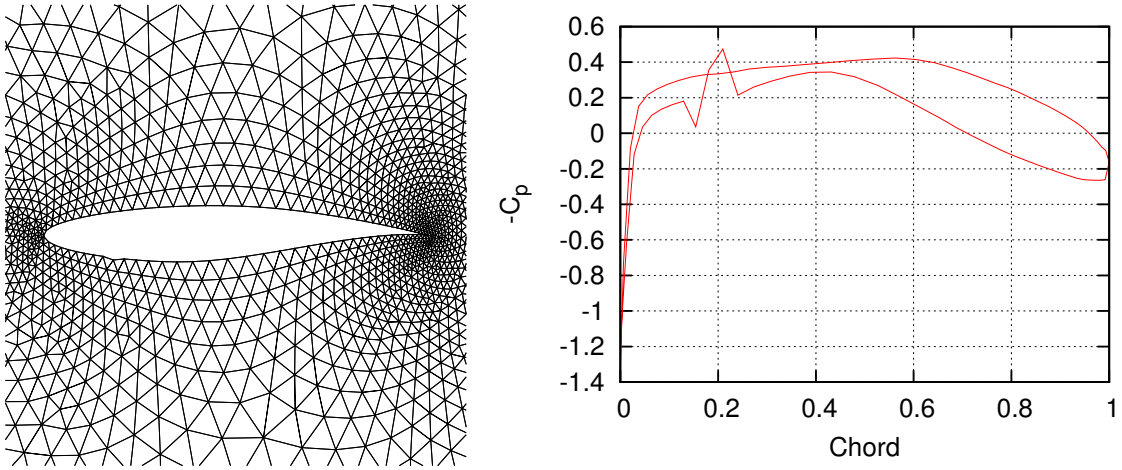


Figure 4.12: Perturbed shape and initial pressure distribution.

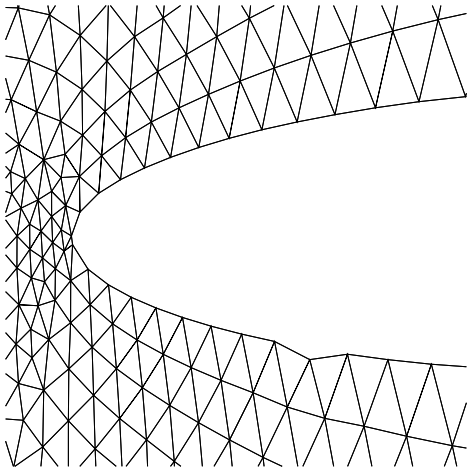


Figure 4.13: Area of the perturbation.

and therefore this first validation of the optimisation algorithm is achieved rapidly.

4.6.2 NACA 0012 to RAE 2822

A more demanding optimization case would be to match an initial shape to a prescribed one. For this purpose, the initial geometry and target geometries used in this example are going to be that of the NACA 0012 and the the RAE 2822 airfoils respectively. The leading and trailing edge are kept fixed. The conditions used are $Ma = 0.43$ and angle of attack $\alpha = 0.0°$. A set of four unstructured grids of triangular elements is used on the NACA 0012, with the finest having 81 surface nodes, 4331 cells and 2212 nodes. The initial shape and pressure distribution are presented in Figure 4.15.
The constant C for the one–shot optimisation (see Section 4.8) is set to 10^5. Also, in every design iteration, one Jacobi smoothing iteration is performed on the design variables and

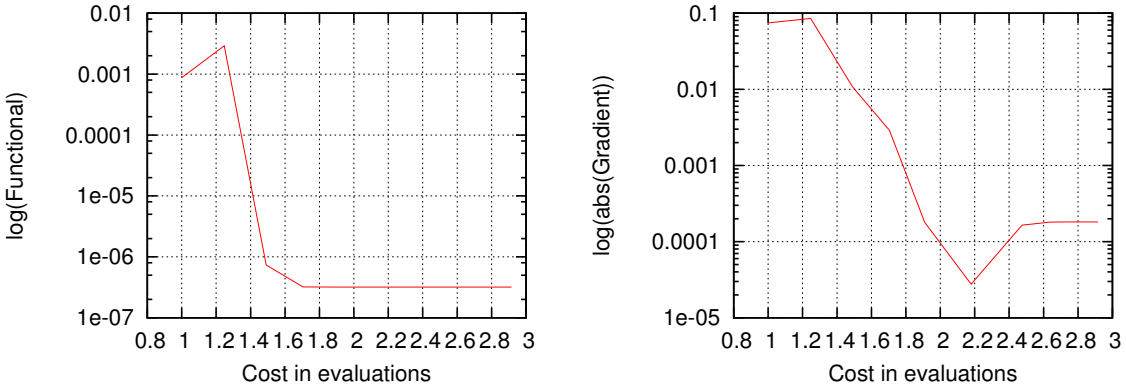


Figure 4.14: Convergence of the functional [top] and the gradient [bottom].

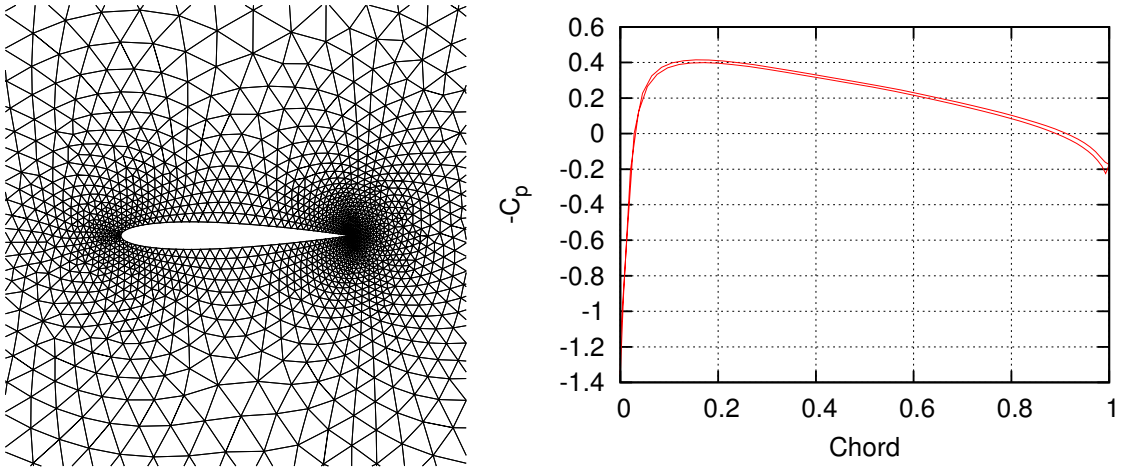


Figure 4.15: Initial shape and pressure distribution.

the gradients, with $\beta = 0.5$ (equation (4.2)). As far as the volume mesh is concerned, ten Jacobi smoothing iterations are performed with $\beta = 0.6$. The functional used in this case is the integral of the L^2_{norm} of the total pressure difference, as in 4.6.1, discussed in Section 2.7.3. The convergence of the functional and the L^2_{norm} of its gradient around the airfoil are presented in Figure 4.16. The initial, target and optimised shapes can be observed in Figure 4.17.

The cost function and the gradients in the pictures above could converge further and therefore the small discrepancy in the shape would not appear but there are two reasons why this does not happen. First, the leading and trailing edges of the two airfoils do not match and second, the wall nodes of the two airfoils do not have the same x coordinates. Since the wall nodes of the NACA airfoil are only allowed to move in the y direction, this constrains the design space. Overall though, the shape change is considered successful and therefore the optimisation algorithm valid.

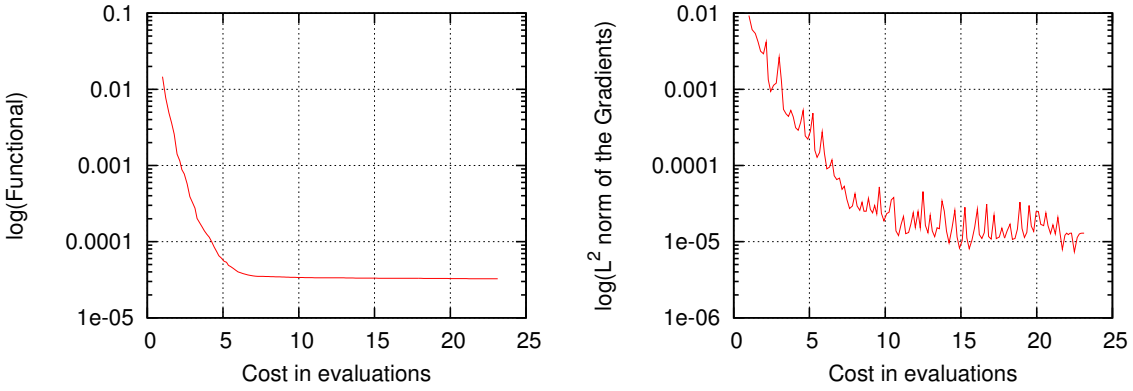


Figure 4.16: Convergence of the functional and the L^2-norm of its gradient.

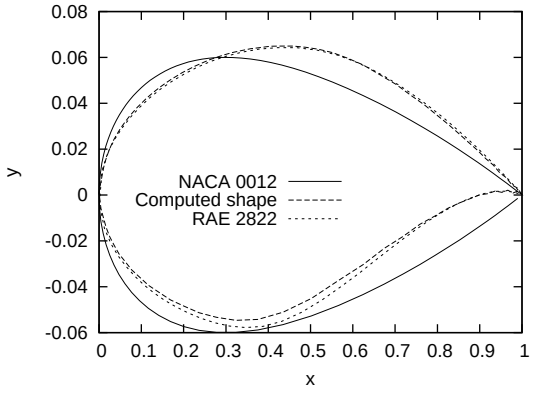


Figure 4.17: NACA 0012 to RAE shape change.

4.6.3 Optimisation verifidation in three dimensions

In order to validate the algorithm in three dimensions as well, the simple and fast to run case of Figure 4.18 was constructed. The case was examined for drag minimisation and the expected result is a flat plate. The leading and trailing edges of the bar were kept fixed in this case and the Euler equations were used. Figure 4.19 demonstrates that the target was achieved and indeed the original shape turns into a flat plate. Figure 4.20 presents the convergence of the cost function and the L^2_{norm} of the gradients.

The previous three subsections were verification cases for the optimisation algorithm, which proved its validity. The following ones are real optimisation cases, in which the final shape is unknown and the optimisation algorithm's effectiveness can only be observed by the performance improvement.

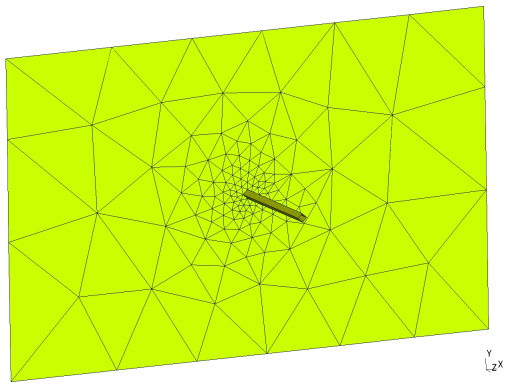


Figure 4.18: Verification case for the optimisation algorithm in three dimensions. The expected optimisation output is a flat plate.

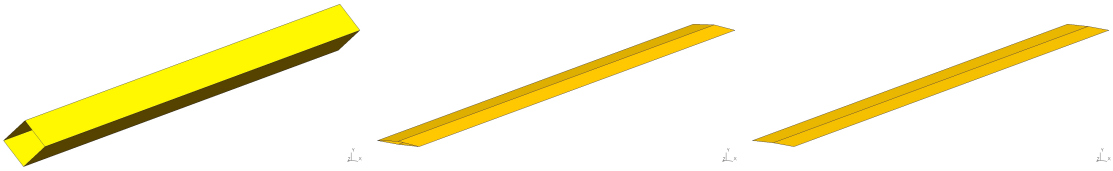


Figure 4.19: Changes of the shape on a 3D optimisation validation case. Initial [left], intermediate [middle] and final [right] shape.

4.6.4 S-Bend duct optimisation

The previous sections presented optimisation verification results for two and three dimensional cases. It is important to examine such cases before examining more general cases. In real life though, the final optimised shape is unknown and the validity of the optimisation algorithm can only be tested by examining the performance improvement. As a case of this form, the S-Bend duct of Figure 4.21 (top) was examined with regards to pressure change. The entire wall was defined as the design surface, while inlet and outlet were kept fixed. With regards to smoothing, one smoothing and ten smoothing iterations were performed on the design surface and the volume grid respectively, with under relaxation factors (β, Section 4.3) 0.5 and 0.7. The one–shot methodology factor C was set to 10^3. In this case, the Euler equations were used, solved with second order accuracy. The convergence of the functional and the L^2_{norm} of the gradients is presented in Figure 4.22.

As it can be observed, although the functional and the gradients converge, the surface mesh is getting rather deformed for a three dimensional case. This was observed after application of various levels of smoothing, from little to a lot of smoothing and for different under relaxation parameter values. Although such a smoothing methodology can be used for Euler flows and relatively coarse meshes, it is not suitable for viscous flows and fine meshes, where the non smooth surfaces cause either divergence in the flow

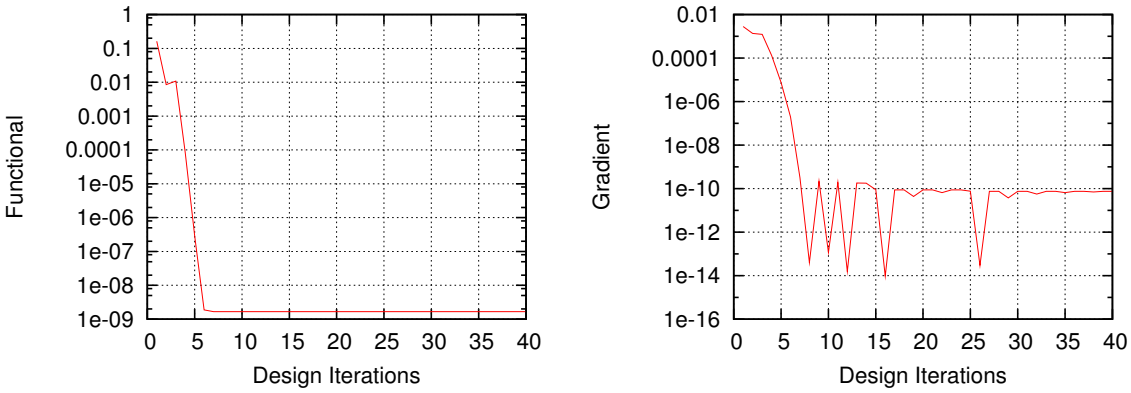


Figure 4.20: Convergence of functional [left] and gradient [right] for a 3D optimisation validation case.

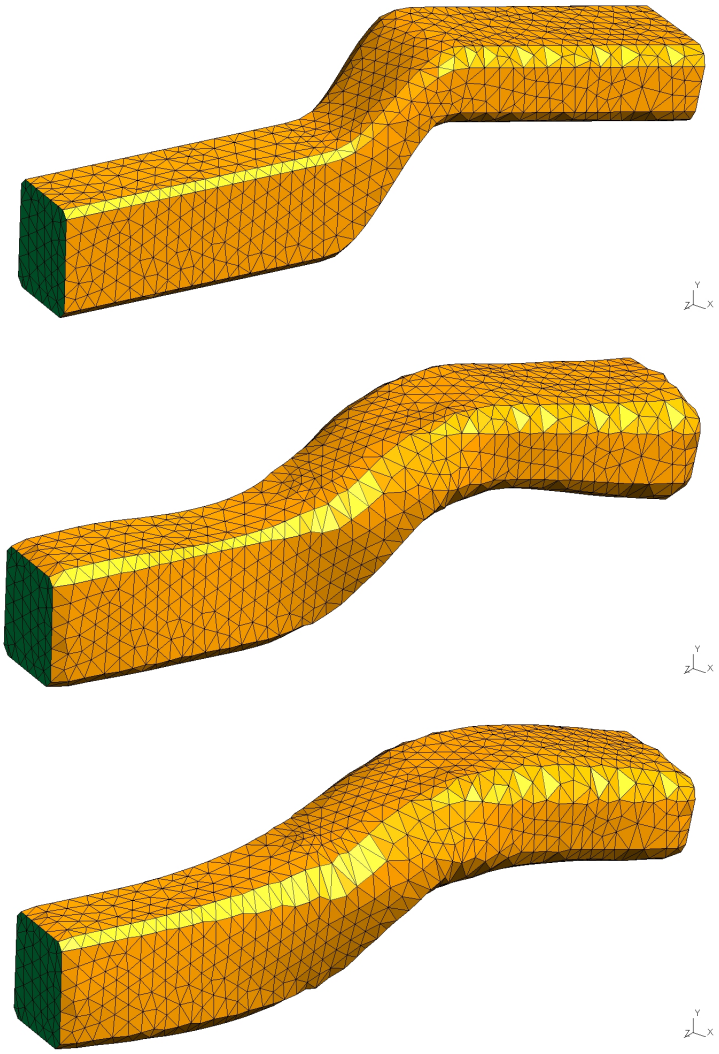


Figure 4.21: Shape change on a S-Bend duct. Initial (top), intermediate (middle) and final (bottom).

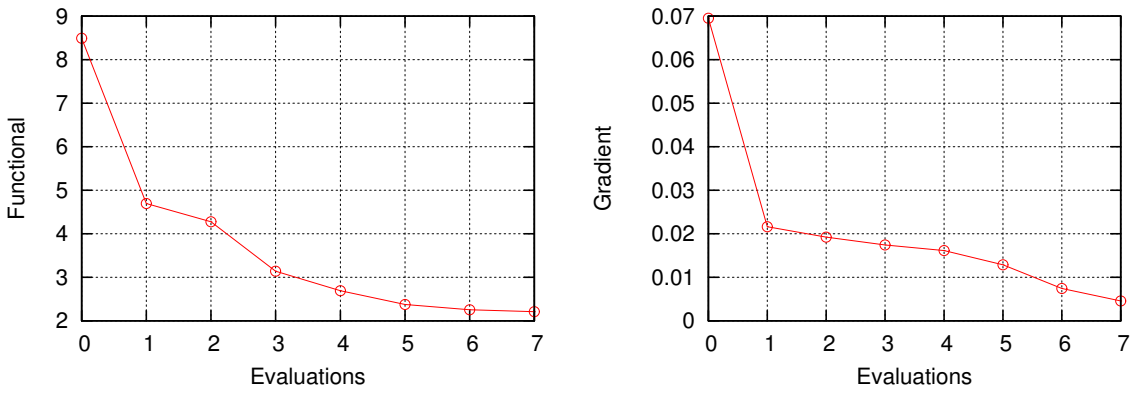


Figure 4.22: Convergence of the functional and the gradients.

or non numbers in the adjoint. A series of viscous cases were examined but the problems above always appeared. This can be bypassed by adopting a more efficient smoothing methodology (e.g. [84]), but this escapes the scope of this research.

4.6.5 Summary

This chapter presented the remaining ingredients needed to construct a gradient based optimisation algorithm. First, the parametrisation was discussed and the node based parametrisation used in this thesis was presented. Second, the necessity for a smoothing algorithm for the design surface and volume mesh (which may also be used for the design) and a simple form of smoothing was presented. Third, the logic behind the scaling of the design variables was outlined and the relative methodology used in the present research was explained. Then, the one-shot methodology was discussed, which accelerates the design convergence. Last, optimisation cases were presented for cases in two and three dimensions. Although the optimisation algorithm was verified and it did improve shapes, it was found the simple form of smoothing is unsuitable for viscous cases and a more sophisticated smoothing algorithm should be implemented in such cases. Considering though that this research is focused on exploring the limits of using AD for aerodynamic optimisation, smoothing escapes the scope of this research. The logic followed though did prove to be valid and is considered successful at this point.

Chapter 5

Hessian computations

5.1 Motivation

The previous chapters discussed the computation of the optimal aerodynamic shape for a given geometry using gradient based methodology and, most efficiently, the adjoint theory. This process was based on first order derivatives, for a specific point of operating conditions. This would then have to be described as *single point* optimisation. Of course, the methodology can be coupled with multi point optimisation techniques, where the cost function J_{total} is constructed by the summation of weighted cost functions $J_{i,\dots,N}$ in N various operating conditions :

$$J_{total} = \sum_{i=1}^{N} w_i J_i \tag{5.1}$$

Apart from that though, it is interesting to examine how robust an optimum solution is. To demonstrate this, let us suppose that a function $f(x)$ may have the form of Figure 5.1. Supposing the definition space of f is that of Figure 5.1, the function has two minimums, a global at x_1 and a local at x_2. At a first glance, x_1 would be the point of interest in an (minimisation) optimisation case, as the value of f is the minimum at $x_1 \forall x$. It can be observed though that, small variations in x would cause large changes in f. On the

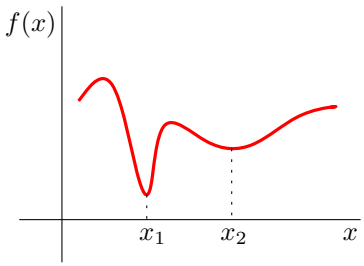


Figure 5.1: Multiple extrema of a function f.

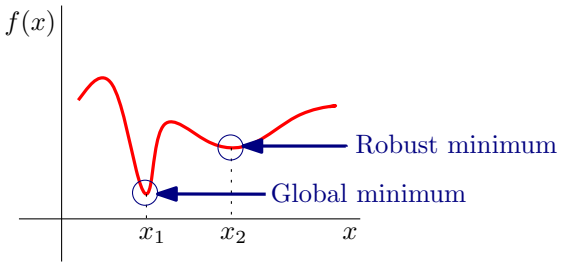


Figure 5.2: Global and robust optima of a function f.

other hand, changes in x in the neighbourhood of x_2 would cause less change in f. In engineering applications, this may in many cases be of greater importance than the global minimum, as it would mean that the optimal design will be able to operate well, even with variations in the operating conditions. Therefore, even if x_1 is the global minimum, the engineer would choose the design of x_2, which would be described as more *robust*, Figure 5.2. Such variations in x would be described as uncertainties and, in real life, they can arise in any point of a product's life, from the mathematical description of the underlying physical models used to design it, to manufacturing and operating. Examples of such uncertainties, e.g. in the field of aeronautical design, could be a varying angle of attack during the different phases of flight, varying atmospheric conditions, tolerances in the manufacturing, geometric variations during flight, geometric variations due to degradation, e.t.c.. All these uncertainties would affect performance and, therefore, their quantification and consideration during the optimisation process would lead to more robust designs [11], than the ones based just on predefined operating conditions. Once the main uncertainties have been identified, quantified and embodied in the optimisation cycle, the cost functional J could be expressed by its expected value μ and its variance σ. Such statistical variables are acquired by integrating in the n-dimensional space of the $\alpha \in \mathbb{R}$ uncertain variables [11, 67] :

$$\mu = \int J(\alpha)\vartheta(\alpha)d\alpha \tag{5.2}$$

$$\sigma^2 = \int J(\alpha)^2\vartheta(\alpha)d\alpha - \mu^2 \tag{5.3}$$

where $\vartheta(\alpha)$ is a probability density function (PDF) for the uncertain variables. Such an integration though is prohibitively expensive to perform, considering the cost of the flow solver. Alternative methods to compute these variables had to be be developed then and the main categories of those are the *Method of Moments* [92], *Monte Carlo* estimates using surrogates and *Polynomial Chaos expansion*.

To use, e.g. a Method of Moments, the variance would need the computation of second

order sensitivity information, as implied by (5.3). But, even without having in mind the methods above, a robust optimum can be acquired by the minimisation of second order sensitivity information (Hessian). So, for example, even when basing the optimisation on first derivatives, a **constraint** could be introduced, enforcing the minimisation of the Hessian along with the first order gradient.

In what follows, the main focus will be the computation and validation of second order derivatives via Algorithmic Differentiation. The first effort to compute Hessians via Automatic Differentiation (AD) was made by Sherman et al [65], giving a detailed description of the underling methodology. Their paper addressed the advantages and disadvantages of each of the four methods of computing a Hessian matrix, that is any combination of the tangent and the adjoint approch, as it will be discussed in Section 5.2. This lead to focusing on the Tangent over Tangent (ToT) and Tangent over Reverse (ToR) approaches. Ghate and Giles [37] made smart use of the adjoint methodology in order to alter the ToT approach and cut the computational cost. Martinelli and Duvigneau [67] employed the alternative ToT of [37] to perform a metamodel-based Monte Carlo for uncertainty estimation in aerodynamics and in [68], Martinelli and Hascoët performed a comparison of the ToT and ToR approaches, by using a simplified model of a CFD code (everything combined within a single function) and, therefore, a brute force application of AD. Last but not least, Rumpfkeil and Mavriplis [93] examined in depth the alternative ToT (which makes use of the adjoint), presenting a detailed analysis that includes most of the components of a modern CFD code.

Apart from those, interesting use of the Hessian matrix has been presented in a couple of papers. First, Zervogiannis, Papadimitiou and Giannakoglou [110],[87] used the Hessian (via continuous adjoint) for Newton and exactly initialised quasi Newton optimisation cases for design convergence acceleration. Also, Naumann et al [83] used the Hessian via AD in Truncated Newton methods.

Effort is therefore being made for efficient computation of the Hessian, in order to be used in optimisation problems, both to examine robustness and design convergence acceleration issues. The present research contributes to this effort by presenting a detailed description of constructing both ToT and ToR algorithms via AD, which are computationally cheaper and able to be initialised. Also, all the various parameters of an CFD aerodynamic optimisation are considered and not only simplified models, as in the rest of the bibliography. The methodology is described in the next sections.

5.2 Methodology

In Chapter 3, the computation of the derivatives of a function $\mathcal{F}$ has been described both for the direct differentiation and discrete adjoint methodology. Similar logic can be followed for the derivation of the second derivative of the function $\mathcal{F}$. The first derivatives can be differentiated by using either direct differentiation or adjoint, resulting in four in total methodologies: Direct differentiation over Direct differentiation, Direct differentiation over Adjoint, Adjoint over Direct differentiation and Adjoint over Adjoint. Using the nomenclature of Automatic Differentiation (see Section 3.4), the respective terminology would be: i.Tangent over tangent (ToT), ii.Tangent over reverse (ToR), iii.Reverse over tangent (RoT) and iv.Reverse over reverse (RoR). The latter nomenclature is going to be frequently used for the rest of the Chapter for simplicity. In [65] certain characteristics of the different approaches are being discussed, which are briefly summarised in Table 5.1. Taking into account those characteristics, the most favourable are the Tangent-over-Tangent (ToT) and the Tangent-over-Reverse (ToR) approaches. In [68], Martinelli and Hascoët compared ToT and ToR by applying brute force AD over a simple example, to propose that ToT performs better for less than 40 design variables but ToR outperforms it for a larger group. Both methodologies are presented in detail in the following sections.

Method	**Cost**	**Note**
ToT	$\sim N_{DV}^2$	Low complexity
ToR	$\sim N_{DV}$	Low complexity
RoT	$\sim N_{DV}$	Low benefit
RoR	$\sim N_{DV}$	High complexity

Table 5.1: Remarks on the different methods to compute second derivatives. N_{DV} is the number of design variables.

5.2.1 Direct differentiation over Direct differetiation

Following this approach, two sequential direct differentiations are applied on a function. In the context of a CFD gradient based optimisation problem (see Chapters 3 and 4) and following this approach, the Hessian of a cost function $J(U, \alpha)$ with respect to the

design variables α (U are the state variables) could be computed by :

$$\begin{aligned}
\frac{dJ}{d\alpha_{i\,k}} &= \frac{\partial J}{\partial \alpha_i} + \frac{\partial J}{\partial U}\frac{dU}{d\alpha_i} \\
\Rightarrow \frac{d^2J}{d\alpha_i d\alpha_k} &= \frac{d}{d\alpha_k}\left(\frac{\partial J}{\partial \alpha_i} + \frac{\partial J}{\partial U}\frac{dU}{d\alpha_i}\right) \\
\Rightarrow \frac{d^2J}{d\alpha_i d\alpha_k} = \frac{\partial^2 J}{\partial \alpha_i \partial \alpha_k} + \frac{\partial^2 J}{\partial U \partial \alpha_i}\frac{dU}{d\alpha_k} &+ \frac{\partial^2 J}{\partial U \partial \alpha_k}\frac{dU}{d\alpha_i} + \frac{\partial}{\partial U}\left(\frac{\partial J}{\partial U}\frac{dU}{d\alpha_i}\right)\frac{dU}{d\alpha_k} + \frac{\partial J}{\partial U}\frac{d^2U}{da_i da_k} \\
\Rightarrow \frac{d^2J}{d\alpha_i d\alpha_k} &= D^2_{i,k}J + \frac{\partial J}{\partial U}\frac{d^2U}{da_i da_k}
\end{aligned} \tag{5.4}$$

For representation simplicity, the differential operator $D^2_{i,k}$, acting on a function $F(U,\alpha)$ has been introduced, where :

$$D^2_{i,k}\mathcal{F} = \frac{\partial^2 \mathcal{F}}{\partial \alpha_i \partial \alpha_k} + \frac{\partial^2 \mathcal{F}}{\partial U \partial \alpha_i}\frac{dU}{d\alpha_k} + \frac{\partial^2 \mathcal{F}}{\partial U \partial \alpha_k}\frac{dU}{d\alpha_i} + \frac{\partial}{\partial U}\left(\frac{\partial \mathcal{F}}{\partial U}\frac{dU}{d\alpha_i}\right)\frac{dU}{d\alpha_k} \tag{5.5}$$

In the equation above, the most expensive term to compute is $d^2U/da_i da_k$, which can be acquired by two direct differentiations of the state equations $R(U,\alpha) = 0$:

$$\begin{aligned}
\frac{d}{d\alpha_k}\left(\frac{dR}{d\alpha_i}\right) &= 0 \\
\Rightarrow \frac{d}{d\alpha_k}\left(\frac{\partial R}{\partial U}\frac{dU}{d\alpha}\right) &= \frac{d}{d\alpha_k}\left(-\frac{\partial R}{\partial \alpha_i}\right) \\
\Rightarrow \frac{\partial^2 R}{\partial U \partial \alpha_k}\frac{dU}{d\alpha_i} + \frac{\partial}{\partial U}\left(\frac{\partial R}{\partial U}\frac{dU}{d\alpha_i}\right)\frac{dU}{d\alpha_k} &+ \frac{\partial R}{\partial U}\frac{d^2U}{d\alpha_i d\alpha_k} = -\frac{\partial^2 R}{\partial \alpha_i \partial \alpha_k} - \frac{\partial^2 R}{\partial U \partial \alpha_i}\frac{dU}{d\alpha_k} \\
\Rightarrow \frac{\partial R}{\partial U}\frac{d^2U}{d\alpha_i d\alpha_k} &= -D^2_{i,k}R
\end{aligned} \tag{5.6}$$

The algorithm for computing the Hessian in such a way, would then have to be :

1. $\forall \alpha_i, i \in [1,n]$, compute and store the solution of the equation : $\frac{\partial R}{\partial U}\frac{dU}{d\alpha_i} = -\frac{\partial R}{\partial \alpha_i}$
2. $\forall \alpha_i, i \in [1,n]$:
   $\forall \alpha_k, k \in [1,i]$:
   (a) Compute the solution to the equation : $\frac{\partial R}{\partial U}\frac{d^2U}{d\alpha_i d\alpha_k} = -D^2_{i,k}R$
   (b) Compute $\frac{d^2J}{d\alpha_i d\alpha_k} = D^2_{i,k} + \frac{\partial J}{\partial U}\frac{d^2U}{d\alpha_i d\alpha_k}$

Provided that the equivalence $\frac{d^2J}{d\alpha_i d\alpha_k} = \frac{d^2J}{d\alpha_k d\alpha_i}$ is satisfied, the algorithm above would have a computational cost of :

$$(N_{DV} + (N^2_{DV} + N_{DV})/2) \cdot N_{it}$$

iterations, where N_{DV} the number of design variables and N_{it} the number of required iterations to solve the tangent or adjoint system (practically the same). Therefore, the cost of the methodology increases quadratically with the number of design variables and its use soon becomes prohibitive for optimisation cases with a large number of design variables.

5.2.2 Alternative Direct differentiation over Direct differetiation

A more efficient way of computing the Hessian in a tangent manner can be acquired by making a few rearrangements in the equations of the previous paragraph. To describe this approach, (5.6) has to be rewritten in the form :

$$(5.6) \Rightarrow \frac{d^2U}{d\alpha_i d\alpha_k} = -\frac{\partial R}{\partial U}^{-1} D^2_{i,k} R \tag{5.7}$$

The term $\frac{d^2U}{d\alpha_i d\alpha_k}$ can then be substituted in (5.4) :

$$(5.4) \overset{(5.7)}{\Rightarrow} \frac{d^2J}{d\alpha_i d\alpha_k} = D^2_{i,k} J - \frac{\partial J}{\partial U}\frac{\partial R}{\partial U}^{-1} D^2_{i,k} R \tag{5.8}$$

In Chapter 3, the derivation of the adjoint equation (3.5)) was presented. That equation can be reformed as :

$$\begin{aligned} \frac{\partial R}{\partial U}^T v &= \frac{\partial J}{\partial U}^T \\ \Rightarrow v &= \frac{\partial R}{\partial U}^{-T} \frac{\partial J}{\partial U}^T \\ \Rightarrow v^T &= \frac{\partial J}{\partial U}\frac{\partial R}{\partial U}^{-1} \end{aligned} \tag{5.9}$$

Then, (5.8) can be written via (5.9) as :

$$(5.8) \overset{(5.9)}{\Rightarrow} \frac{d^2J}{d\alpha_i d\alpha_k} = D^2_{i,k} J - v^T D^2_{i,k} R \tag{5.10}$$

In this way, the algorithm to compute the Hessian changes into :

1. Solve : $\frac{\partial R}{\partial U}^T v = \frac{\partial J}{\partial U}^T$
2. $\forall \alpha_i, i \in [1, n]$, solve and store the solution of : $\frac{\partial R}{\partial U}\frac{dU}{d\alpha_i} = -\frac{\partial R}{\partial \alpha_i}$
3. $\forall \alpha_i, i \in [1, n]$:

$\forall \alpha_k, k \in [1, i]$:
Compute $\frac{d^2 J}{d\alpha_i d\alpha_k} = D^2_{i,k} + \frac{\partial J}{\partial U}\frac{d^2 U}{d\alpha_i d\alpha_k}$

The computational cost of this alternative algorithm is :

$$(1 + N_{DV}) \cdot N_{it} + (N^2_{DV} + N_{DV})/2$$

The relation to the number of design variables is still quadratic, but the number of required iterations is reduced. This alternative tangent over tangent approach, which though makes use of the adjoint, is cheaper that the pure ToT methodology but still expensive for large cases with great number of design variables. It should be emphasized that, although this methodology does make use of the adjoint, it is actually in a Direct differentiation over Direct differentiation logic. The use of the adjoint is just a mathematical rearrangement.

5.2.3 Direct differantiation over Adjoint

A more promising methodology for the computation of the Hessian is Direct differentiation over Adjoint approach. The second derivative information in this case is derived by differentiating the first derivative (computed via adjoint) in a direct differentiation manner. In the context of a CFD optimisation problem, as in the previous paragraph, this is described in the following equations :

$$\begin{aligned}
\frac{dJ}{d\alpha_i}^T &= \frac{\partial J}{\partial \alpha_i}^T + \frac{\partial R}{\partial \alpha_i}^T v \\
\Rightarrow \frac{d}{d\alpha_k}\left(\frac{dJ}{d\alpha_i}^T\right) &= \frac{d}{d\alpha_k}\left(\frac{\partial J}{\partial \alpha_i}^T\right) + \frac{d}{d\alpha_k}\left(\frac{\partial R}{\partial \alpha_i}^T v\right) \\
\Rightarrow \frac{d}{d\alpha_k}\left(\frac{dJ}{d\alpha_i}^T\right) &= \frac{\partial}{\partial \alpha_k}\left(\frac{\partial J}{\partial \alpha_i}^T\right) + \frac{\partial}{\partial U}\left(\frac{\partial J}{\partial \alpha_i}^T\right)\frac{dU}{d\alpha_k} + \frac{\partial}{\partial \alpha_k}\left(\frac{\partial R}{\partial \alpha_i}^T v\right) + \\
&\quad \frac{\partial}{\partial U}\left(\frac{\partial R}{\partial \alpha_i}^T v\right)\frac{dU}{d\alpha_k} - \frac{\partial R}{\partial \alpha_i}^T \frac{dv}{d\alpha_k} \\
\Rightarrow \frac{d^2 J}{d\alpha_i d\alpha_k} &= \dot{\bar{J}}_\alpha - \dot{\bar{R}}_\alpha - \frac{\partial R}{\partial \alpha_i}^T \frac{dv}{d\alpha_k}
\end{aligned} \tag{5.11}$$

For simplicity, the terms $\dot{\bar{J}}_\alpha$ and $\dot{\bar{R}}_\alpha$ have been introduced, which are given by :

$$\dot{\bar{J}}_\alpha = \frac{\partial}{\partial \alpha_k}\left(\frac{\partial J}{\partial \alpha_i}^T\right) + \frac{\partial}{\partial U}\left(\frac{\partial J}{\partial \alpha_i}^T\right)\frac{dU}{d\alpha_k} \tag{5.12}$$

$$\dot{\bar{R}}_\alpha = \frac{\partial}{\partial \alpha_k}\left(\frac{\partial R}{\partial \alpha_i}^T v\right) + \frac{\partial}{\partial U}\left(\frac{\partial R}{\partial \alpha_i}^T v\right)\frac{dU}{d\alpha_k} \tag{5.13}$$

The most expensive term to compute in (5.11) is $dv/d\alpha_k$, which represents the sensitivity of the adjoint variables with respect to the design variables. This term can be computed by applying direct differentiation on the adjoint equation (3.5) to acquire (5.14) :

$$\begin{aligned}
\frac{d}{d\alpha_k}\left(\frac{\partial R}{\partial U}^T v\right) &= \frac{d}{d\alpha_k}\left(\frac{\partial J}{\partial U}^T\right) \\
\frac{\partial}{\partial \alpha_k}\left(\frac{\partial R}{\partial U}^T v\right) + \frac{\partial}{\partial U}\left(\frac{\partial R}{\partial U}^T v\right)\frac{dU}{d\alpha_k} + \frac{\partial R}{\partial U}^T \frac{dv}{d\alpha_k} &= \frac{\partial}{\partial \alpha_k}\left(\frac{\partial J}{\partial U}^T\right) + \frac{\partial}{\partial U}\left(\frac{\partial J}{\partial U}^T\right)\frac{dU}{d\alpha_k} \\
\frac{\partial R}{\partial U}^T \frac{dv}{d\alpha_k} = \left[\frac{\partial}{\partial \alpha_k}\left(\frac{\partial J}{\partial U}^T\right) + \frac{\partial}{\partial U}\left(\frac{\partial J}{\partial U}^T\right)\frac{dU}{d\alpha_k}\right] &- \left[\frac{\partial}{\partial \alpha_k}\left(\frac{\partial R}{\partial U}^T v\right) + \frac{\partial}{\partial U}\left(\frac{\partial R}{\partial U}^T v\right)\frac{dU}{d\alpha_k}\right] \\
\Rightarrow \frac{\partial R}{\partial U}^T \frac{dv}{d\alpha_k} &= \dot{\bar{J}}_U - \dot{\bar{R}}_U \qquad (5.14)
\end{aligned}$$

where, similarly, the terms $\dot{\bar{J}}_U$ and $\dot{\bar{R}}_U$ are given by :

$$\dot{\bar{J}}_U = \frac{\partial}{\partial \alpha_k}\left(\frac{\partial J}{\partial U}^T\right) + \frac{\partial}{\partial U}\left(\frac{\partial J}{\partial U}^T\right)\frac{dU}{d\alpha_k} \qquad (5.15)$$

$$\dot{\bar{R}}_U = \frac{\partial}{\partial \alpha_k}\left(\frac{\partial R}{\partial U}^T v\right) + \frac{\partial}{\partial U}\left(\frac{\partial R}{\partial U}^T v\right)\frac{dU}{d\alpha_k} \qquad (5.16)$$

The algorithm to compute the Hessian in way is 6:

1. Solve the primal.
2. Compute the adjoint solution from : $\frac{dJ}{d\alpha_i} = \frac{\partial J}{\partial \alpha_i}^T + v^T f$.
3. $\forall \alpha_i, i \in [1, n]$:
   (a) Compute the tangent solution from : $\frac{\partial R}{\partial U}\frac{dU}{d\alpha} = -\frac{\partial R}{\partial \alpha_i}$
   (b) Compute the terms $\dot{\bar{J}}_\alpha$, $\dot{\bar{R}}_\alpha$, $\dot{\bar{J}}_U$ and $\dot{\bar{R}}_U$.
   (c) Solve $\frac{\partial R}{\partial U}^T \frac{dv}{d\alpha_k} = \dot{\bar{J}}_U - \dot{\bar{R}}_U$.
   (d) Compute $\frac{d^2 J}{d\alpha_i d\alpha_k} = \dot{\bar{J}}_\alpha - \dot{\bar{R}}_\alpha - \frac{\partial R}{\partial \alpha_i}^T \frac{d}{d\alpha_k}\left(\frac{dU}{d\alpha_i}^T\right)$

The computational cost of the algorithm above is :

$$N_{it} + 2 \cdot N_{it} \cdot N_{DV}$$

iterations. This implies a linear relation to the number of design variables and therefore more suitable for real-world industrial optimisation cases than ToT. For this reason, the ToR approach is the one of most interest in this research. Once the methodologies above have been implemented in the framework of a software, these results have do be verified. This process is described in the next section.

5.3 Second derivatives computation via AD

The previous paragraphs presented the methodologies of most interest for the derivation of second derivatives of a cost function in the context of a CFD optimisation problem. The implementation of those in a software can be performed either by hand or by Automatic Differentiation. The present chapter aims to explore the latter and present a novel methodology for the fully automated generation of second order sensitivity code.

5.3.1 Preparation and application of double differentiation

At this stage, it is supposed that all the requirements discussed in Section 3.4.1 are met, that is the syntax of the source code is in accordance with the capabilities of the AD tool and blanking directives have been introduced. This means that the source code can be parsed by the AD tool for the first differentiation. The second differentiation is applied on the AD generated sensitivity code, which will be suitable for automatic differentiation (since it is written out by the tool itself). Therefore, no extra effort is needed for these two steps, regarding the second differentiation.

A part of the preparation for the second differentiation that needs attention is the identification of the dependent and independent variables, so that the equivalent directions to the AD tool are implemented correctly. In the case of Tangent over Tangent approach (ToT), the equivalent to the Direct differentiation over Direct differentiation, the logic of differentiation does not change, compared to the first one: independents remain independents and the same for dependants. If for example, if the function $f(x, y)$ is computed by the routine "`subroutine functionF (x,y,f)`", its first and second derivatives are going to be respectively computed by the following AD generated routines :

```
subroutine functionF_d (x,xd,y,yd,f,fd)
subroutine functionF_d_d (x,,xd,xdd,y,yd,ydd,f,fd,fdd)
```

were `d` and `dd` denote first and second differentiation in forward mode. In all the above routines, [`x,xd,xdd`] and [`y,yd,ydd`] are inputs and `[f,fd,fdd]` outputs. This means that the logic of information flow of the primal routine does not change in either the first or second differentiation. This shall be used as a guideline, when writing the differentiation directions to the AD tool. In this case, the tool shall be called for the first differentiation with [`x,y`] inputs and [`f`] output and for the second [`x,xd,y,yd`] and [`fd`] likewise.

On the other hand, this is not the case in the Tangent over Reverse approach, the equivalent of AD to Direct differentiation over Adjoint. The first differentiation of the primal code version of $f(x, y)$ in reverse mode would result in the routine "`subroutine`

`functionF_b (x,xb,y,yb,f,fb)`", where [`x,y,fb`] are inputs and [`f,xb,yb`] outputs. The logic of the flow information has now been reversed and input symbols became outputs and vice versa. The second differentiation in tangent mode would generate "`subroutine functionF_b_d (x,xb,xd,xbd,y,yb,yd,ybd,f,fb,fd,fbd)`" and maintain the logic of information flow of the adjoint, thus [`x,xd,y,yd,fb,fbd`] would be inputs and [`xb,xbd,yb,ybd,fd`] outputs. Therefore, the AD tool would have to be called with [`x,y`] and [`f`] as inputs and output respectively for the first differentiation but with inputs [`x,y,fb`] and outputs [`xb,yb,f`] for the second.

After this logic is understood, the rules to the AD tool for the differentiation can be written. For the automation of this procedure, it is again proposed to use a *Makefile*, as in Chapter 3. This will contain the preprocessing commands (see Section 3.4.1), the rules for the first and second differentiation, the definitions of the code to be differentiated and the call to the AD tool. For example, the differentiation directionsl implemented for **mgOpt** are presented in Figure 5.3. The call to the tool can be performed in the same way as Figure 3.3.

5.3.2 Embodying in the non–differentiated souce code

Once the methodology has been implemented, the sensitivity code computing second derivatives can be generated via AD. The next step into using this code in the relative software is to identify the terms involved in the methodology of Section 5.2 in the AD generated second derivative routines and link them to the the source code accordingly. To demonstrate this in the context of a CFD solver, two functions of high importance are considered, the flux/residual routine and the cost function, as they are involved in the computation of the complicated terms are $D^2_{i,k}R$ and $D^2_{i,k}J$ (see Section 5.2. For simplicity, these are considered here to be functions of the design variables α and the state $U(\alpha)$: $R : (\alpha, U(\alpha)) \mapsto R(\alpha, U(\alpha))$ and $J : (\alpha, U(\alpha)) \mapsto J(\alpha, U(\alpha))$ and have the form (only arguments of most importance are presented) :

```
subroutine residuals (a,U,R)
subroutine functional (a,U,J)
```

Using the ToT approach, the twice differentiated versions of these functions would be :

$$\texttt{subroutine residuals\_d\_d (a,}\overset{\delta\alpha}{\downarrow}\texttt{ad,}\overset{\delta^2\alpha}{\downarrow}\texttt{add,U,}\overset{\frac{dU}{d\alpha}}{\downarrow}\texttt{Ud, Udd ,R,Rd, Rdd)}\ \underset{D^2_{i,k}R}{\downarrow}$$

$$\texttt{subroutine functional\_d\_d (a,}\overset{\delta\alpha}{\downarrow}\texttt{ad,}\overset{\delta^2\alpha}{\downarrow}\texttt{add,U,}\overset{\frac{dU}{d\alpha}}{\downarrow}\texttt{Ud, Udd ,J,Jd,Jdd)}\ \underset{D^2_{i,k}J}{\downarrow}$$

```
# SECOND DERIVATIVES
ifneq ($(AD_MODE),)
ad2:=$(shell \
  if [[ $(AD_MODE) -eq 11 || $(AD_MODE) -eq 21 ]]; \
    then echo "d"; \
  elif [[ $(AD_MODE) -eq 12 || $(AD_MODE) -eq 22 ]]; \
    then echo "b"; \
  fi)
endif

ifeq ($(ad2),d)
  ifeq ($(ad1),d) # Tangent over Tangent (ToT)
    m3:=m05_residual$(CP1_FS)_$(ad1)%
    m5:=m05_residual$(CP2_FS)_$(ad1)%
    m6:=m02_clcdcp$(CP1_FS)_$(ad1)%
    m8:=m06_bruteforce_$(ad1)%
    m11:=m03_adkeep_$(ad1)%
    AD2_CMD:=
    AD2_CMD:=$(m1)residual_$(ad1)(cv res)\(res$(ad1))
    AD2_CMD+=$(m5)residual$(CP2_FS)_$(ad1)(cvin cv)\(res$(ad1))
    AD2_CMD+=$(m9)residual$(CP1_FS)_$(ad1)(cv)\(res$(ad1))
    AD2_CMD+=$(m2)calclift_$(ad1)(q q$(ad1) alRadInf)\(cl$(ad1))
    AD2_CMD+=$(m6)calclift$(CP1_FS)_$(ad1)(q)\(cl$(ad1))
    AD2_CMD+=$(m7)calclift$(CP2_FS)_$(ad1)(alRadInf)\(cl$(ad1))
    AD2_CMD+=$(m4)initialisation_$(ad1)(alp)(cv$(ad1) cvin$(ad1))
    AD2_CMD+=$(m11)keep_grid_t_elem_$(ad1)
    AD2_CMD+=$(m8)brute_$(ad1)(x_vrt)\(cl$(ad1) cv$(ad1))
  endif
  ifeq ($(ad1),b) # Tangent over Reverse (ToR)
    m1:=m05_residual$(CP1_FS)_$(ad1)%
    m2:=m05_residual$(CP2_FS)_$(ad1)%
    m3:=m02_clcdcp$(CP1_FS)_$(ad1)%
    m4:=m03_adkeep_$(ad1)%
    m5:=m06_bruteforce_$(ad1)%
    m6:=m02_clcdcp_$(ad1)%
    m7:=m04_timestep$(CP1_FS)_$(ad1)%
    m8:=m02_smoothing_$(ad1)%
    m9:=m03_norm77_$(ad1)%
    AD2_CMD:=
    AD2_CMD+=$(m1)residual$(CP1_FS)(cv res)\(res)
    AD2_CMD+=$(m2)residual$(CP2_FS)(vol x_vrt nds_iedg nds_bvrt x_fce x_bfce res)\(res)
    AD2_CMD+=$(m1)residual$(CP1_FS)_$(ad1)(cv cv$(ad1) res$(ad1))\(cv cv$(ad1))
    AD2_CMD+=$(m2)residual$(CP2_FS)_$(ad1)(vol nds_iedg nds_bvrt x_vrt x_fce x_bfce dv \
             vol$(ad1) nds_iedg$(ad1) nds_bvrt$(ad1) x_vrt$(ad1) x_fce$(ad1) x_bfce$(ad1) \
             dv$(ad1) res$(ad1))\(vol$(ad1) nds_iedg$(ad1) nds_bvrt$(ad1) x_vrt$(ad1) \
             x_fce$(ad1) x_bfce$(ad1) dv$(ad1) res$(ad1))
    AD2_CMD+=$(m4)keep_grid_t_elem
    AD2_CMD+=$(m5)brute_$(ad1)(x_vrt)\(x_vrt$(ad1))
    AD2_CMD+=$(m6)calclift(bwt q)\(cl)
    AD2_CMD+=$(m7)update_cv$(CP1_FS)_$(ad1)(res cv0 cv$(ad1))\(cv0$(ad1) cv$(ad1) res$(ad1))
    AD2_CMD+=$(m8)smoothen_3D(dVar)\(xMod)
    AD2_CMD+=$(m9)norm77(x egNrm bwt midFF midBFF)\(egNrm bWt midFF midBFF x)
    AD2_CMD+=$(m6)lift_drag_3D(q bwt)\(cl cd)
    AD2_CMD+=$(m6)lift_drag_3D_$(ad1)(q bwt)\(q$(ad1) bwt$(ad1))
    AD2_CMD+=$(m9)norm77_$(ad1)(x egNrm bwt)\(x$(ad1))
    AD2_CMD+=$(m8)smoothen_3D_$(ad1)(dVar)\(dVar$(ad1))
    AD2_CMD+=$(m5)brute3D_$(ad1)(dvar)\(dvar$(ad1))
    AD2_CMD+=$(m6)intPress(q bwt)\(func)
  endif
endif
```

Figure 5.3: Rules for the second differentiation implemented in the *Makefile*. *AD_MODE* is provided by the user as a part of the *make* command and determines the type of differentiation. *CP_FS* determines the suffixes for the files having multiple copies.

where the inputs and outputs of most interest are presented. $\delta\alpha$ and $\delta^2\alpha$ are the 1^{st} and 2^{nd} order perturbations of the design variables α. The most important terms of the equations presented in Section 5.2.1 are marked as inputs (on top of the argument list) and outputs (below the argument list) in the differentiated routines above.

Using the ToR approach on the same primal functions, the twice differentiated routines would have the form :

$$\texttt{subroutine residuals\_b\_d (a,}\overset{\delta\alpha}{\underset{\downarrow}{\texttt{ad}}}\texttt{,ab, }\underset{\downarrow}{\texttt{abd}}\texttt{, U,}\overset{\frac{dU}{d\alpha}}{\underset{\downarrow}{\texttt{Ud}}}\texttt{,Ub, }\underset{\downarrow}{\texttt{Udd}}\texttt{, R,}\overset{v}{\underset{\downarrow}{\texttt{Rb}}}\texttt{)}$$
$$\dot{\bar{R}}_\alpha \qquad\qquad \dot{\bar{R}}_U$$

$$\texttt{subroutine functional\_b\_d (a,}\overset{\delta\alpha}{\underset{\downarrow}{\texttt{ad}}}\texttt{,ab, }\underset{\downarrow}{\texttt{abd}}\texttt{, U,}\overset{\frac{dU}{d\alpha}}{\underset{\downarrow}{\texttt{Ud}}}\texttt{,Ub, }\underset{\downarrow}{\texttt{Udd}}\texttt{, J,Jb)}$$
$$\dot{\bar{J}}_\alpha \qquad\qquad \dot{\bar{J}}_U$$

where $dU/d\alpha$ the tangent solution, v the adjoint solution and $\delta\alpha$ the perturbation of the design variables α. The terms involved in the equations of Section 5.2.3 are presented as inputs and outputs in a similar way to ToT.

Having identified the terms of the mathematical model in the differentiated routines, the latter can be called in an appropriate manner from the source code. Before they can by used though, issues of linking and compiling have to be bypassed. These are discussed in the next section.

5.4 Linking & compiling

The issues with linking and compiling that arose in for the first differentiation are again present for the second differentiation and they can be dealt with in a similar way. Apart from those though, there is an additional problem in the case of double differentiation, which is related with AD tools using stack operations (e.g. PUSH/POP [47] functions for *Tapenade* [101]). After the first differentiation, the generated code will include such functions in the case of reverse accumulation. During the second differentiation, the calls to these functions are going to be differentiated and therefore a differential version of them has to be provided for the compiler to be able to link and compile. Figure 5.4 presents such and example for the case of ToR. A similar problem would occur if the Reverse over Reverse (RoR) methodology was examined. If the source code of these functions is provided by the AD tool developers, then it can be added to the code to be differentiated. If not, then the relative binaries have to be provided. In the case of *Tapenade* [101], which is used in this thesis, the source code of such functions and their differentiated versions are provided. This though was only a relatively recent addition to the tool and might not be the case for other AD tools. Also, although the later were provided, there was a number of programming bugs involved, which the author reported to the developers and which are now corrected. All the above indicate that the procedure might not be implemented straightforwardly and that it is not guaranteed to work. In the case of **mgOpt** though, these problems were dealt with and therefore second derivative computations were enabled both for ToT and ToR.

5.5 Verification of second derivatives

After the generation of the relative sensitivity code, the second derivatives computed via AD need to be verified. For this, a similar approach to Section 3.5 is followed. Gradients computed via ToT and ToR are verified between themselves and their magnitude is confirmed by Finite Differences. It shall be noted though that FD can be noisy for

```
DO i=1,n_vrt
  CALL PUSHREAL8ARRAY(dv(:, i), 9)
  CALL PUSHREAL8ARRAY(cv(:, i), n_dx + 3)
  CALL CALC_DV(n_dx, cv(:, i), rgamm, rcpgas, &
               rvis, pr, prt, dv(:, i,))
END DO
```

```
DO i=1,n_vrt
  CALL PUSHREAL8ARRAY_D(dv(:, i), dvd(:, i), 9)
  CALL PUSHREAL8ARRAY_D(cv(:, i), cvd(:, i), n_dx + 3)
  CALL CALC_DV_D(n_dx, cv(:, i), cvd(:, i), rgamm, rcpgas, &
                 rvis, pr, prt, dv(:, i), dvd(:, i))
END DO
```

```
DO i=n_vrt,1,-1
  CALL POPREAL8ARRAY(cv(:, i), n_dx + 3)
  CALL POPREAL8ARRAY(dv(:, i), 9)
  CALL CALC_DV_B(n_dx, cv(:, i), cvb(:, i), &
     rgamm, rcpgas, rvis, pr,prt,dv(:, i),dvb(:, i))
END DO
```

```
DO i=n_vrt,1,-1
  CALL POPREAL8ARRAY_D(cv(:, i), cvd(:, i), n_dx + 3)
  CALL POPREAL8ARRAY_D(dv(:, i), dvd(:, i), 9)
  CALL CALC_DV_B_D(n_dx, cv(:, i), cvd(:, i), cvb(:, i), cvbd(:, i), &
       rgamm, rcpgas, rvis, pr, prt, dv(:, i), dvb(:, i),dvbd(:, i))
END DO
```

Figure 5.4: Stack operations occurring from the first reverse differentiation and there differential equivalents after the second forward differentiation.

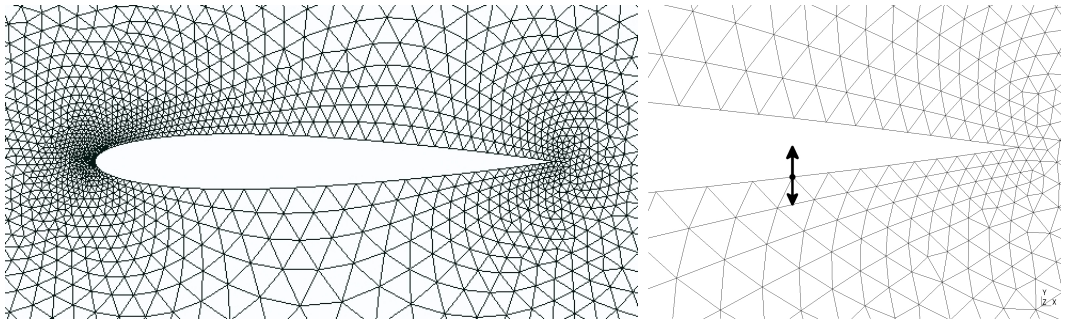


Figure 5.5: Naca 0012 and perturbation mode for second derivative computation.

second derivatives and therefore not be able to provide an estimation to the gradients. On the other hand, ToT and ToR should match to machine precision [46].

To demonstrate this methodology, an example has been constructed, where the sensitivity of lift on a 2D airfoil NACA 0012 (Fig. 5.5) is computed with respect to a single nodal perturbation on the surface. This example has been selected on purpose, as the visualisation of second order sensitivities in the computational domain in three dimensional cases can be difficult. The farfield conditions used were Mach number $Ma = 0.4$ and angle of attack $a = 2°$. The gradient values computed via FD, tangent, adjoint, ToT and ToR are compared in Table 5.2 for the $AUSM^+_{up}$ flux. It can be observed that AD gradients match to machine precision and their magnitude is verified by FD. The accuracy of FD though is limited and it decreases even further for second order accurate computations. Figures 5.6 and 5.7 make a visual comparison for the gradient fields of y-velocity. Only minor differences can be observed, verifying that the gradients via AD are computed correctly.

Despite the fact that the second derivative was verified for the $AUSM^+_{up}$ scheme, problems were encountered for the Roe scheme. Although the first gradients were verified for this scheme (also see Section 3.5), only the second derivatives via the ToT approach were computed correctly. In the case of ToR, the magnitude of the computed gradients did not match the ones from ToT and FD. Although the source code was differentiated

$AUSM^{+}_{up}$	Method	1^{st} Order Accuracy	2^{nd} Order Accuracy
First	FD	**0.453321**42571	**0.619166**53538
Derivative	Tangent	**0.453321292403**	**0.619166824022**
	Adjoint	**0.453321292403**	**0.619166824022**
Second	FD	**138.3578**951408	**142.2**067719136
Derivative	ToT	**138.3578906648**	**142.2184126185**
	ToR	**138.3578906648**	**142.2184126185**

Table 5.2: Gradient comparison on NACA 0012, using *AUSM+ up*.

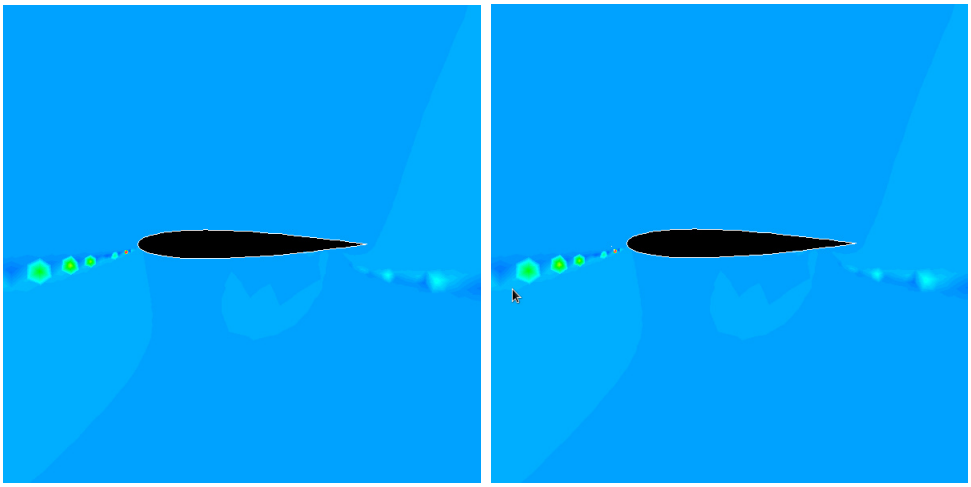


Figure 5.6: First order accurate sensitivity fields of y-velocity via FD [left] and ToT [right].

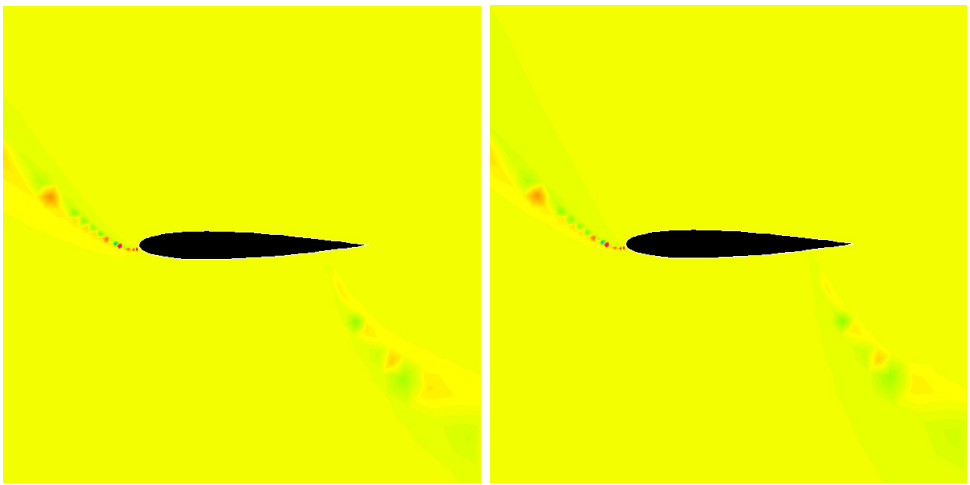


Figure 5.7: Second order accurate sensitivity fields of y-velocity via FD [left] and ToT [right].

ROE	Method	1^{st} Order Accuracy	2^{nd} Order Accuracy
First	FD	**0.488457**739739	**0.623140**574551
Derivative	Tangent	**0.488457**833838	**0.623140**717437
Second	FD	**181**.7805383996	**168.0**491124034
Derivative	ToT	**181**.8151423224	**168.0**571788088

Table 5.3: Gradient comparison on NACA 0012, using *Roe*'s flux.

correctly and the AD generated routines were invoked in the right way, the gradients via ToR failed to match the correct value. Also, the value of the FD or ToT computed second derivative using the Roe scheme was similar to the ones via $AUSM^{+}_{up}$ but did differ to a noticeable extend. This was not the case for first derivatives. Based on all the above, this discrepancy lead to the assumption that the Roe scheme might not be suitable for double differentiation, especially using the ToR methodology.

5.6 Performance improvement

The previous sections presented the methodology for the derivation of sensitivity code that computes second derivatives via AD and the relative verification. Apart from these though, it is also important to accelerate the computations of such information. As discussed in Section 3.6, there are several ways, in which AD can be applied. The black, grey and white box use of AD revealed a increasing performance but also complexity and effort. The white box approach proposed though was proved to have the highest performance and also enabled the adjoint to be hot–started (initialised with a pre–existing solution). It was the most complex one but this was restrained only to the initial preparation. After the latter, the process was automated via *Makefile*. Apart from this, coupling of the adjoint with the primal multi–grid and flux pre–conditioner was performed, which resulted in further performance improvement.

In the same logic, this section presents the effort to accelerate the computations of second derivatives via AD. The white box approach application of AD for second derivatives is presented and the relative logic is also coupled with the primal multi–grid.

5.6.1 Selective use of AD and hand assembly

The main features of a CFD code include metrics computations (normal vectors, volumes, etc.), fixed point iterations and cost functions. Summarising those in a sample code, the relative functionalities would be in the order of Table 5.4. The symbols '$\rightarrow$' and '$\leftarrow$'

```
call metrics ( →X, ←Mtrx )
do nIter = 1,mIt
           call residual ( →U, →Mtrx, ←R )
           call update ( →R, ⇌U )
end do
call cost_function ( →U, →Mtrx, ←J )
```

Table 5.4: Representation of a sample flow solver.

```
call metrics_d_d (→X,→X_d,→X_d_d,←Mtrx,←Mtrx_d,←Mtrx_d_d )
do nIter = 1,mIt
  call residual_d_d (→U,→U_d,→U_d_d,→Mtrx,→Mtrx_d,→Mtrx_d_d,
           ←R,←R_d,←R_d_d )
  call update_d_d ( →R,→R_d,→R_d_d,⇌U,⇌U_d,⇌U_d_d)
end do
call cost_function_d_d (→U,→U,→U,→U,→Mtrx,←J )
```

Table 5.5: Representation of brute–force ToT applied on the code of Table 5.4

imply inputs and outputs respectively, `X` are design variables and `Mtrx` represents the metrics. Using source code transformation AD and the black or grey box approach (see Section 3.6), the generated sensitivity code using the ToT or the ToR methodology would have the form of Table 5.5 or Table 5.6 respectively. In practice, the algorithm of Tables 5.6 would not have such a clean form but would incorporate taping (stack) operations (e.g. push/pop, see [101]), which increase the memory and runtime of computations.

```
call metrics_d (→X,→X_d,←Mtrx,←Mtrx_d)
do nIter = 1,mIt
  call residual_d (→U,→U_d,→Mtrx,→Mtrx_d,←R,←R_d)
  call update_d ( →R,→R_d,⇌U,⇌U_d)
end do
call cost_function_b_d (→U,→U_d,←U_b,←U_b_d,→Mtrx,→Mtrx_d,
            ←Mtrx_b,←Mtrx_b_d,←J,→J_b)
do nIter = mIt,1,-1
  call update_b_d (→R,→R_d,←R_b,←R_b_d,→U,→U_b,→U_b_d )
  call residual_b_d (→U,→U_d,←U_b,←U_b_d,→Mtrx,→Mtrx_d,
            ←Mtrx_b,←Mtrx_b_d,←R,←R_d,→R_b,→R_b_d)
end do
call metrics_b_d (→X,→X_d,←X_b,←X_b_d,←Mtrx,←Mtrx_d,
           →Mtrx_b,→Mtrx_b_d, )
```

Table 5.6: Representation of brute–force ToR applied on the code of Table 5.4

Instead of such a brute–force approach to using AD, the logic described in Section 3.6 and [19] for first derivatives can be used for second derivatives, in order to acquire better performance, without unnecessary stack operations and computations. In this way, AD is

```
call metrics_d_d (→X,→X_d,→X_d_d,←Mtrx,←Mtrx_d,←Mtrx_d_d )
call residual_MTRX_d_d (→Mtrx,→Mtrx_d,→Mtrx_d_d,←R,←f,←DR )
do nIter = 1,mIt
  call residual_U_d_d (→U,→U_d,→U_d_d,←R,←R_d,←R_d_d )
  R_d = R_d - f
  R_d_d = R_d_d - DR
  call update_d_d ( →R,→R_d,→R_d_d,⇌U,⇌U_d,⇌U_d_d)
end do
call cost_function_d_d (→U,→U,→U,→U,→Mtrx,←J )
```

Table 5.7: Representation of hand assembled ToT with selective differentiation applied on the primal code of Table 5.4

used in a better way, that increases the performance of the code. For this, the derivative code would have to be hand assembled and various functions differentiated multiple times with respect to different arguments each time (e.g. the residual/flux computation function in a similar logic to Section 3.6. For example, the algorithms of Table 5.5 would take the form of Table 5.7. Similarly, the algorithm of Table 5.6 can take the form of Table 5.8. Note that the arguments `U_b` and `U_b_d` have been switched with `R_b` and `R_b_d` respectively in Table 5.8, in order to maintain the logic of information flow of the algorithm. Through this *white box* application of AD for second derivatives, the solution of the second order systems can be initialised by selecting a *hot–start*, avoiding starting from the same point each time (as in the black-box approach). Furthermore, the runtime is decreased by restraining the taping operations. An example of the latter is presented for ToR in Table 5.10. Last but not least, the form of the assembled algorithms above, enables the use of the original multi–grid, as it will be described in the next subsection.

On the other hand though, there is also a number of disadvantages. First, the initial preparation of the differentiation requires the maximum amount of effort and can be time consuming. This includes selecting the right code to differentiate, understanding which functions will be differentiated multiple time and with respect to which arguments, implementing and testing the differentiation rules, hand assembling the algorithms and implementing the interfaces for automatic embodying of the differentiated routines to the rest of the code. Although this is a complicated procedure, the linking of the differentiated code to the rest of the source code can be fully automated via *Makefile* by following the methodology of Section 3.6.1, once the initial necessary preparations have been made. Apart from this, the verification process is more time consuming and can be quite complex when debugging issues arise.

```
call metrics_d (→X,→X_d,←Mtrx,←Mtrx_d)
call residual_Mtrx_d (→Mtrx,→Mtrx_d,←R,←f(=R_d))
do nIter = 1,mIt
  call residual_d (→U,→U_d,←R,←R_d)
  R_d = R_d - f
  call update_d ( →R,→R_d,⇌U,⇌U_d)
end do
call cost_function_b_d (→U,→U_d,←g,←w,→Mtrx,→Mtrx_d,
            ←Mtrx_b,←Mtrx_b_d,←J,→J_b)
do nIter = mIt,1,-1
  call residual_b_d (→U,→U_d,←R_b,←R_b_d,→Mtrx,→Mtrx_d,
            ←Mtrx_b,←Mtrx_b_d,←R,←R_d,→U_b,→U_b_d)
  R_b = R_b - g
  R_b_d = R_b_d - w
  call update_b_d (→R,→R_d,←U_b,←U_b_d,→U,→R_b,→R_b_d )
end do
call metrics_b_d (→X,→X_d,←X_b,←X_b_d,←Mtrx,←Mtrx_d,
           →Mtrx_b,→Mtrx_b_d, )
```

Table 5.8: Representation of hand assembled ToR with selective differentiation applied on the primal code of Table 5.4

Methodology	**Runtime [sec]**
Brute force ToR	2944.4360
Proposed ToR	2387.4854
Improvement	**19%**

Table 5.9: Performance acceleration using the proposed ToR methodology.

5.6.2 Use of the primal multi–grid

Once the *white box* use of AD described in the previous paragraph is implemented, coupling of the methodology with a multi–grid methodology can be examined. In this research effort was made to use the primal geometric multi–grid (Section 2.8.1) instead of a differentiated one. The restriction/prolongation operations are performed in the same way as the primal, but the second derivative systems' variables and residuals (equations (5.6) and (5.14)) are used instead. Using the nomenclature of Section 2.8.1 and the FAS methodology [15], the multi–grid operations for the ToT are :

1. *Smooth errors on the finer level.*

$$\left[\frac{d^2U}{da_i da_k}\right]^n = \left[\frac{d^2U}{da_i da_k}\right]^n + \Lambda\left[[-D^2_{i,k}R]^h - \mathcal{A}^h\left[\frac{d^2U}{da_i da_k}\right]^n\right] \tag{5.17}$$

2. *Add the ToT source term to the finer level ToT residuals.*

$$\ddot{R}^h = \ddot{R}^h - D^2_{i,k} R \tag{5.18}$$

3. *Transfer the ToT solution and residuals to the coarser level.*

$$\left[\frac{d^2U}{da_i da_k}\right]^H = I_h^{H,U}\left[\frac{d^2U}{da_i da_k}\right]^h \text{ and } \ddot{R}^H = I_h^{H,\ddot{R}}\ddot{R}^h \tag{5.19}$$

4. *Prolong the coarser level ToT correction to the finer level.*
   After smoothing on the coarsest level :

$$\left[\frac{d^2U}{da_i da_k}\right]^H = \left[\frac{d^2U}{da_i da_k}\right]^H + \Lambda\left[[-D^2_{i,k}R]^H - \mathcal{A}^H\left[\frac{d^2U}{da_i da_k}\right]^H\right] \tag{5.20}$$

   the coarser level correction is prolonged to the finer level :

$$\left[\frac{d^2U}{da_i da_k}\right]^h = \left[\frac{d^2U}{da_i da_k}\right]^h + I_H^h\left[\left[\frac{d^2U}{da_i da_k}\right]^H - I_h^{H,U}\left[\frac{d^2U}{da_i da_k}\right]^h\right] \tag{5.21}$$

Following the same logic, the multi–grid operations in the case of ToR are :

1. *Smooth errors on the finer level.*

$$\left[\frac{dv}{d\alpha_k}\right]^n = \left[\frac{dv}{d\alpha_k}\right]^n + \Lambda\left[[\dot{\bar{J}}_U - \dot{\bar{R}}_U]^h - [\mathcal{A}^T]^h\left[\frac{dv}{d\alpha_k}\right]^n\right] \tag{5.22}$$

2. *Add the ToR source term to the finer level ToR residuals.*

$$\dot{\bar{R}}^h = \dot{\bar{R}}^h + [\dot{\bar{J}}_U - \dot{\bar{R}}_U]^h \tag{5.23}$$

3. *Transfer the ToR solution and residuals to the coarser level.*

$$\left[\frac{dv}{d\alpha_k}\right]^H = I_h^{H,U}\left[\frac{dv}{d\alpha_k}\right]^h \text{ and } \dot{\bar{R}}^H = I_h^{H,\dot{\bar{R}}}\dot{\bar{R}}^h \tag{5.24}$$

4. *Prolong the coarser level ToR correction to the finer level.*
   After smoothing on the coarsest level :

$$\left[\frac{dv}{d\alpha_k}\right]^H = \left[\frac{dv}{d\alpha_k}\right]^H + \Lambda\left[[\dot{\bar{J}}_U - \dot{\bar{R}}_U]^H - [\mathcal{A}^T]^H\left[\frac{dv}{d\alpha_k}\right]^H\right] \tag{5.25}$$

the coarser level correction is prolonged to the finer level :

$$\left[\frac{dv}{d\alpha_k}\right]^h = \left[\frac{dv}{d\alpha_k}\right]^h + I_H^h \left[\left[\frac{dv}{d\alpha_k}\right]^H - I_h^{H,U}\left[\frac{dv}{d\alpha_k}\right]^h\right] \tag{5.26}$$

As in the case of the primal, tangent and adjoint, multi–grid can accelerate the converge of the ToT and ToR systems. An example for ToR on the case of Section 3.6.2 (with the same multi–grid configuration) is presented in Table 5.10.

Methodology	**Runtime [sec]**	Improvement
Black box AD	2944.4360	-
White box AD	2387.4854	19%
White box AD with MG	707.0922	76%

Table 5.10: Performance acceleration using the primal multi–grid on ToR.

5.7 Summary

This chapter presented the motivation behind the derivation of second derivatives and described in detail two approaches to the latter: Direct Differentiation over Direct Differentiation and Direct Differentiation over adjoint. In the present thesis, the author aimed to explore the relative methodologies via Automatic Differentiation (AD) and propose a state–of–the–art methodology for source code preparation, application of AD, post–code generation linking and compiling and automation of the entire process (after the initial manual preparations). Although a brute–force approach to the use of AD for second derivatives was examined, the findings of the research revealed that a selectively differentiated source code and hand assembled sensitivity algorithm outperforms the first. Also, the methodology was coupled with the primal geometric multi–grid of the developed software **mgOpt**, revealing further performance acceleration. Last but not least, issues of code maintenance were discussed and the logic behind the via AD and *Makefile* automated sensitivity software maintenance was explained.

Concluding this chapter, it shall be mentioned that the entire logic above has been derived, implemented, embodied, validated and automated in the relative research software **mgOpt**. The main focus in the case of the present research was to explore the derivation of second derivatives via AD, automate it and improve the performance, targets which have been met. The absence of parallelism on the software at the present time prohibits the application of the methodology on a real world large case. It will be available for use though to the future researchers of the research group, enhancing further research in the

relative area. The main focus in the present research was to expand the automated use of AD in CFD codes and implement scripts that maintain the code efficiently, as it will be discussed further in the following chapter. For this reason, there will be no further application of second derivatives in this thesis.

Chapter 6

Epilogue

6.1 The thesis in a few words

The previous chapters presented the research that was carried out by the author during his doctorate study years. In what follows, the summary of the entire thesis is outlined.

Chapter 1 consisted a pre-phase to the chapters that followed, presenting the motivation for the presented research. The evolution in the field of fluid mechanics and optimisation was presented, which revealed the need for automatic optimisation and design processes based on reverse engineering and especially the adjoint methodology. Given such an automatic procedure, the time and cost of the engineering design process can be minimised and the relative constructions would perform most efficiently.

In Chapter 2, the basic features of the CFD code **mgOpt** further developed throughout the writer's research have been presented. The space discretisation and the methods use for solving the flow equations have been described in detail. Furthermore, solution accuracy issues have been discussed and the way second order accurate solutions are computed was explained, including mathematical explanation of the scheme and the flux limiters used. The two methodologies used for performance acceleration, multi–grid and pre–conditioning, have also been presented. Finally, a number of flow cases have been examined, which validated the results of the solver before the the implementation of the sensitivity solvers, as well as a few that were going to be used in the later chapters for sensitivity computations and optimisation.

Moreover, the adjoint methodology was exploited in Chapter 3. First, a basic reference to direct differentiation (or tangent in Algorithmic Differentiation terminology) was examined, in order to form a basis for the adjoint methodology. Then, the discrete adjoint approach was presented, which revealed the advantages of using such a methodology for optimisation cases with a large number of design variables. After presenting the theory behind the approach, the methodology used in this thesis for the derivation of the adjoint

via Automatic Differentiation was presented. The difficulties were addressed and ways to bypass them were proposed. A novel methodology for the complete automation of the generation of the adjoint (and tangent) sensitivity code was detailed, which makes use of advanced scripting and AD. All these were applied on high level Fortran 90/95, which uses pointers, derived data types, modules etc. This was another novelty of the present research as AD has been applied in CFD on the simpler syntax of Fortran 77 until now and the relevant literature was lacking the use of AD on more complex syntaxes. The thesis outlines the difficulties and could constitute a how–to guide to future researchers in this area. The relative work has also contributed to the further development of the AD tool *Tapenade* [101] by reporting bugs and suggestions. Further in the chapter, the verification of the computed gradients was performed. Later in the chapter, issues of performance acceleration of the adjoint via AD were addressed. The hand assembly of the sensitivity code was found to outperform the brute–force use of AD and the methodology in which this can be performed was described. Apart from this, the hand assembled adjoint from AD was also coupled with the non differentiated multi–grid methodology as well as the bock Jacobi pre–conditioner, which was not differentiated but based on the primal. Last, examples of sensitivities were presented on various geometries and the ways in which these can be used have been discussed.

Chapter 4 was devoted to the use of gradients computed via the adjoint methodology from AD for aerodynamic shape optimisation. The parametrisation was discussed and the node based logic was presented. Issues of smoothing and design scaling were outlined along with the relative methodology used is this thesis. Apart from these, the one–shot methodology was introduced to the optimisation algorithm, in order to accelerate the design convergence and reduce the computational cost. Optimisation validation cases were examined for both two and three dimensional cases before the algorithm was used in unknown optimal shape cases.

Chapter 5 expanded the methodology presented until then and explored the computation of higher order derivatives via AD. The Tangent over Tangent and Tangent over Reverse methodologies were presented, before the issues of using AD for this process on a modern programming language were addressed. The methodology used to bypass upcoming problems was described and a novel methodology to completely automate the generation of the higher order sensitivity code was proposed. Afterwards, the computed higher sensitivities were verified. Also, a state–of–the art hand assembled form of the higher order sensitivity code was presented, which outperformed the brute–force derived one. Last, the hand–assembled sensitivity algorithm was coupled with the non-differentiated multi–grid to achieve even further performance acceleration. This had also not been presented in literature until now.

The chapters above were also accompanied by two appendixes, which are included in the end of the thesis. Although not a main part of the latter, these appendixes presented the logic behind algorithmic differentiation and an example of AD and also discussed issues of multi-capability software generation.

6.2 Presentations and publications

Any research should not only explore new areas but also share the knowledge acquired with other people. Following this logic, the findings in this thesis were presented in a number of scientific conferences and workshops, which are summarised bellow.

1. EUROGEN 2009
   Evolutionary and Deterministic Methods for Design, Optimization and Control with Applications to Industrial and Societal Problems
   Crakow, Poland, 15–17 June 2009
   Presentation: *Parametrisation and smoothing for multigrid design optimisation*

2. BFG 2009
   14th Belgian-French-German Conference on Optimisation
   Leuven, Belgium, 14–18 September 2009
   Presentation: *Adjoint-based Design in the Automotive Industry*

3. ICFD 2010
   Numerical Methods for Fluid Dynamics
   Reading, United Kingdom, 12–15 April 2010
   Presentation: *Discrete adjoint CFD codes and fixed-point iterations using Automatic Differentiation*

4. ECCOMAS CFD 2010
   Fifth European Conference on Computational Fluid Dynamics
   Lisbon, Portugal, 14–17 June 2010
   Presentation: *Pseudo-time stepping for adjoint CFD codes from Automatic Differentiation*

5. FLOWHEAD workshop 2010
   Workshop on industrial design optimisation for fluid flow
   Varna, Bulgaria, 22–24 September 2010
   Presentation: *Hessian computations in CFD via Automatic Differentiation*

6. 8^{th} ASMO UK
   Conference on Engineering Design Optimisation
   London, United Kingdom, 8–9 July 2010
   Presentation: *Adjoint based flow optimisation*

7. ECCOMAS 2011
   CFD & Optimisation
   Antalya, Turkey, 23-25 May 2011
   Presentation: *Second derivatives in CFD via Automatic Differentiation and validation*

8. FLOWHEAD & ESI 2012
   Conference on Industrial Design Optimisation for Fluid Flow
   Munich, Germany, 28—29 March 2012
   Presentation: *Discrete adjoint CFD solver for first and second derivatives via Automatic Differentiation*

9. ASMO/ISSMO UK 2012
   Conference on Engineering Design Optimization
   University College Cork, Cork, Ireland, 5–6 July 2012 Presentation: *Discrete adjoint solvers in industrial design*

Also, main parts of this work have been documented in the following journal and conference proceedings papers :

1. Christakopoulos F., Jones D. and Müller J.-D.
   Pseudo-timestepping and verification for automatic differentiation derived CFD codes
   Computers & Fluids, Volume 46, Issue 1, July 2011, Pages 174-179, ISSN 0045-7930, doi:10.1016/j.compfluid.2011.01.039

2. Jones D., Christakopoulos F and Müller J.-D.
   Preparation and assembly of discrete adjoint CFD codes
   Computers & Fluids, Volume 46, Issue 1, July 2011, Pages 282-286, ISSN 0045-7930, doi:10.1016/j.compfluid.2011.01.042

3. Yu G, Jones D., Christakopoulos F. and Müller J.-D.
   CAD-based shape optimisation using adjoint sensitivities
   Computers & Fluids, Volume 46, Issue 1, July 2011, Pages 512-516, ISSN 0045-7930, doi:10.1016/j.compfluid.2011.01.043

4. Christakopoulos F., Müller J.-D. and Jones D.
   Timestepping for adjoint CFD codes from Automatic Differentiation
   J. C. F. Pereira and A. Sequeira, ECCOMAS CFD 2010 Proceedings

5. Christakopoulos F., Müller J.-D. and Jones D.
   Second derivatives in CFD via Automatic Differentiation and validation
   CFD & Optimisation conference proceedings, ECCOMAS 2011, No:55

6. Christakopoulos F. and Müller J.-D.
   Accelerated and initialisation enabled adjoint based Hessian computations in CFD via Automatic Differentiation
   Computer Methods in Applied Mechanics and Engineering, written/to be submitted

6.3 Discussion

In this thesis it was found that the process of sensitivity code generation can be completely automated both for first and second order derivatives and at the same time have a competitive performance. This process should be first manually prepared though.

In order to reach those findings, source code transformation Automatic Differentiation (AD) and advanced scripting were used. The combination of the two proved to be a competitive alternative to the hand derived, error prone equivalent methodologies. Once the initial preparations have been made, any version of the sensitivity code of a CFD solver (tangent, adjoint, etc.) can be automatically generated, incorporating any changes, improvements and additions made to the latter. This can increase productivity and enhance the enrichment of CFD solvers and gradient based optimisation software, while at the same time the computed gradients are guaranteed correspond correctly to the primal code. Apart from the use of the methodology for first derivatives and optimisation, its use for the derivation of second derivatives can provide accurate information available for use for robust optimisation. Although this methodology does indeed lead to automated sensitivity sensitivity code generation, it does require a considerable amount of time for the hand assembly and a very good understanding not only of the physical and mathematical models but also of algorithmic differentiation. This is essential during the debugging process in order to bypass problems. Therefore, although automatic in the end, the method is in the beginning time consuming and effort demanding.

Apart from the benefits of the methodology proposed though, there is a number of potential weaknesses as well. First, the parallelisation of sensitivity codes from AD is still a issue that has not been dealt with in depth at the present time. Second, the extension of

the methodology in unsteady flows could present increased memory requirements. Last, there is dependency on Automatic Differentiation tools, which is not always desirable, either because of financial reasons or because the latter are still under development not robust/mature enough to deal with all the upcoming shortcomings.

During the research, there was an unexpected finding regarding the differentiability of the Roe flux function. It was observed that the twice differentiated function would provide inaccurate results, which were orders of magnitude different from the expected ones. Despite the effort to compute the correct gradients, no error was found in the implementation and this lead to the conclusion that maybe the Roe flux is not appropriate for double differentiation.

The present study contributes to the field of aerodynamic shape optimisation by presenting the way first and second order sensitivity codes can be fully automatically generated, while having a competitive performance at the same time. Apart from this, was applied on a modern language and not a older one with simpler syntax, which was generally the case in literature studies. It has been proved that such a procedure is indeed feasible.

Closing this discussion the author would like to make two recommendations for future research on this field. First, unsteady flows would be an interesting field for the adjoint methodology from AD. Second, fluidstructure interaction problems have not been examined in the context of adjoint before and they could benefit from gradient based optimisation methods as well.

6.4 Funding Acknowledgement

The current research work would not be possible without financial support and the writer would like to express his appreciation to the two main sponsors. The first one, covering the living cost, is the European research project, of which the research is part of :

FlowHead
Fluid Optimisation Workflows for Highly Effective Automotive Development Processes
Funded by the European Commission (7^{th} Framework Program) under theme :
SST.2007-RTD-1: Competitive product development (February 2009 to January 2012)
http://flowhead.sems.qmul.ac.uk/

The second, covering the cost of tuition fees, is a scholarship by the *School of Engineering and Materials Science, Queen Mary, University of London, E1 4NS, London, United Kingdom, http://www.sems.qmul.ac.uk/.*

Appendix A

An example of algorithmic differentiation

This appendix uses a simple function to present the methodology of algorithmic differentiation, both in a direct differentiation (Section 3.2) and adjoint (Section 3.3) manner. The Automatic Differentiation (AD) tool *Tapenade* [101] is also used to demonstrate the automatic derivation of the sensitivity code that computes the gradient of that function. For this, suppose a function $f(x, y), f : \mathbb{R} \mapsto \mathbb{R}$, which is described by :

$$f(x, y) = x \sin y \tag{A.1}$$

Following differentiation rules, the analytic derivatives of f with respect to each of the independent variables x and y would be :

$$\frac{\partial f}{\partial x} = \sin y \quad \text{and} \quad \frac{\partial f}{\partial y} = x \cos y \tag{A.2}$$

This would be a direct differentiation of f. The equivalent procedure in algorithmic differentiation is called *forward* accumulation and its first step is to break f into its most elemental parts, forming the *elemental list*, Table A.1. Then, the elemental entities are differentiated and assembled using the chain rule so as to form the derivative of

Elemental part	**Equivalent evaluation**
$p_1 = x$	x
$p_2 = y$	y
$p_3 = \sin p_2$	$\sin y$
$p_4 = p_1 \cdot p3$	$x \sin y$

Table A.1: Elemental list of f

Forward code list	Linearisation on x	Linearisation on y
∇p_1	1	0
∇p_2	0	1
$\nabla p_3 = \cos p_2 \cdot \nabla p_2$	0	$\cos y$
$\nabla p_4 = \nabla p_1 \cdot p_3 + p_1 \cdot \nabla p_3 \cdot$	$\sin y$	$x \cos y$

Table A.2: Forward code list example.

the function with respect to each of the independent variables. This forms the *forward list*, Table A.2. It can be observed that the last operations of the forward list are the calculations of $\partial f/\partial x$ and $\partial f/\partial y$ respectively, which are identical to equation (A.2).

The same derivatives can be computed in an adjoint manner, which is referred to as *reverse* accumulation in algorithmic differentiation. The procedure in this case would be again to form the code list and then differentiate the last elemental entity with respect to all of them, from last to first (reverse). This would form the *reverse* list, Table A.3. It can be observed that $\frac{\partial p_4}{\partial p_1}$ and $\frac{\partial p_4}{\partial p_2}$ are the derivatives $\partial f/\partial x$ and $\partial f/\partial y$ respectively (equation (A.2)). The benefit of the reverse accumulation is that all the derivatives of a function can be computed in a single parse, as demonstrated in this small example.

Reverse list	Equivalent evaluation
$\frac{\partial p_4}{\partial p_4} = 1$	1
$\frac{\partial p_4}{\partial p_3} = p_1$	x
$\frac{\partial p_4}{\partial p_2} = \frac{\partial p_4}{\partial p_3}\frac{\partial p_3}{\partial p_2} = p_1 \cdot \cos p_2$	$x \cdot \cos y$
$\frac{\partial p_4}{\partial p_1} = p_3$	$\sin y$

Table A.3: Reverse code list example

After presenting the algorithmic differentiation of the simple example above, the use of an Automatic Differentiation (AD) tool can be demonstrated. This will apply the methodology on the source code (of a computing language) that computes the function f and output the equivalent sensitivity code. For this purpose the source code transformation AD tool *Tapenade* [101] is used, which is also the one used throughout this thesis. Using Fortran 90/95 syntax [18], the function f could be computed by the subroutine of Figure A.1. Calling the AD tool in forward mode along with stating the inputs/outputs of the function would produce the tangent linear code of Figure A.2. Each of the derivatives $\partial f/\partial x$ and $\partial f/\partial y$ can now be computed by calling the sensitivity code with different arguments, Figure A.3. The subscript–like symbols "D" and "d" denote tangent sensitivities forward mode differentiation (see [47]). The sensitivity code would have to be invoked as many times as the number of independent variables.

```
subroutine simple_function (x,y,f)
  real(8),intent(in):: x,y
  real(8),intent(out):: f
  f=x*sin(y)
end subroutine
```

Figure A.1: The simple function f coded in Fortran 90/95.

```
!  Generated by TAPENADE     (INRIA, Tropics team)
!  Tapenade 3.6 (r4165) - 21 sep 2011 20:54
!
!  Differentiation of simple_function in forward (tangent) mode:
!   variations   of useful results: f
!   with respect to varying inputs: x y
!   RW status of diff variables: f:out x:in y:in
SUBROUTINE SIMPLE_FUNCTION_D(x, xd, y, yd, f, fd)
  IMPLICIT NONE
  REAL*8, INTENT(IN) :: x, y
  REAL*8, INTENT(IN) :: xd, yd
  REAL*8, INTENT(OUT) :: f
  REAL*8, INTENT(OUT) :: fd
  INTRINSIC SIN
  fd = xd*SIN(y) + x*yd*COS(y)
  f = x*SIN(y)
END SUBROUTINE SIMPLE_FUNCTION_D
```

Figure A.2: Generated tangent code via AD.

```
!Computation of df/dx
xd=1; yd=0
call SIMPLE_FUNCTION_D (x,xd,y,yd,f,fd)
df_x=fd ! df/dx

!Computation of df/dy
xd=0; yd=1
call SIMPLE_FUNCTION_D (x,xd,y,yd,f,fd)
df_y=fd ! df/dy
```

Figure A.3: Partial derivatives via forward mode AD.

```
!  Generated by TAPENADE     (INRIA, Tropics team)
!  Tapenade 3.6 (r4165) - 21 sep 2011 20:54
!
!  Differentiation of simple_function in reverse (adjoint) mode:
!   gradient     of useful results: f
!   with respect to varying inputs: f x y
!   RW status of diff variables: f:in-zero x:out y:out
SUBROUTINE SIMPLE_FUNCTION_B(x, xb, y, yb, f, fb)
  IMPLICIT NONE
  REAL*8, INTENT(IN) :: x, y
  REAL*8 :: xb, yb
  REAL*8 :: f
  REAL*8 :: fb
  INTRINSIC SIN
  xb = SIN(y)*fb
  yb = x*COS(y)*fb
  fb = 0.0_8
END SUBROUTINE SIMPLE_FUNCTION_B
```

Figure A.4: Generated adjoint code via AD.

```
!Computation of df/dx and df/dy
fb=1
call SIMPLE_FUNCTION_B (x,xb,y,yb,f,fb)
df_x=xb ! df/dx
df_y=yb ! df/dy
```

Figure A.5: Partial derivatives via reverse mode AD.

Calling the AD tool in reverse mode, the sensitivity code produced would have the form of Figure A.4. This code can now only be invoked once to provide the sensitivities of the function with respect to all the independent variables, Figure A.5. The subscript–like symbols "B" and "b" denote reverse mode differentiation (see [47]).

Appendix B

Sensitivity software maintanance automation

A challenge that software developers have to deal with is maintenance. In the case of sensitivity based CFD optimisation programs, this issue can hinder development and improvement, as addition to the source code would main that the equivalent derivative code has to be changed. In this thesis several types of derivative codes have been discussed and the the ones that compute first derivatives in direct differentiation and adjoint manner (Chapter 3) as well as second derivatives using the ToT and ToR methodologies (Chapter 5). The maintenance of software containing all of those, like **mgOpt**, the relative research software of this thesis, could be time wise prohibitive. A novel methodology was presented though, which can maintain the code automatically, after a number of initial preparations. This is making use of Automatic Differentiation (AD) and advanced *Makefile* scripting (Chapters 3 and 5). Once the methodology presented in those two chapters is implemented, sofware of various capabilities can be generated. For example, **mgOpt** can be compiled with the *make* commands of Table B.1 to generate versions of the same software with differencing capabilities. This table can be further enriched using the methodologies described in the thesis, for example for the case of RoR or for even higher derivatives.

***make* command**	Capability of the generated software
make	Flow computation
make AD_MODE=1	Flow and tangent computation
make AD_MODE=2	Flow and adjoint adjoint computation
make AD_MODE=11	Flow, tangent and ToT computation
make AD_MODE=21	Flow, tangent, adjoint and ToR computation

Table B.1: Various *make* commands that control the generation of sensitivity code in **mgOpt**.

```
#ifdef INCLUDE_IN_ADJOINT

  adjoint: if (tm%pgm_task(1:2).eq.'FG' .or. tm%pgm_task(1:7).eq.'ADJOINT' .or.
               tm%pgm_task(1:3).eq.'2ND'.or. tm%pgm_task(1:2).eq.'FD') then
           ...
           end if adjoint

#endif

#ifdef INCLUDE_TANGENT

  tangent: if (tm%pgm_task(1:3)=='2ND' .or. (tm%pgm_task(1:2)=='FD' .and. FD_save==1)) then
           ...
           end if tangent

#endif

#ifdef INCLUDE_IN_TOR

  ToR: if (tm%pgm_task(1:3)=='2ND' .or. (tm%pgm_task(1:2)=='FD' .and. FD_save==1)) then
       ...
       end if ToR

#endif
```

Figure B.1: Definitions of algorithm blanking pre–processing directives.

For this process to be fully automatic though, an additional step is required. In a program that is able to compute derivatives using the methodologies above, the relative algorithms will be calling differentiated functions. If those functions were absent, the source code would be unable to compile. To understand this better, **mgOpt** and the commands of Table B.1 are considered. If it is desired that the executable is only able to compute the flow and therefore no function is differentiated, the program would be unable to compile because the algorithms that compute the tangent, adjoint, ToT, ToR etc. would call the relative differentiated routines but the latter would be absent. Similar examples can be made for any case of Table B.1. To bypass this problem and make the *make* commands above possible, source code blanking pre–processor directives need to be introduced in the source code. These will hide certain parts of the source code to the compiler, depending to the *make* command. Examples of such directives are presented in Figure B.1. The variables `INCLUDE_IN_ADJOINT`, `INCLUDE_IN_TANGENT`, `INCLUDE_IN_TOR` etc. are defined in the *Makefile*, according to the *make* command given at compile time, Figure B.2. This is the last pre–processing step that concludes the necessary operations for a sensitivity computing program to be fully automated.

```
# MAKE AND SRC BLANKING
ifeq ($(AD_MODE),11)
  CPP_DER2:=-D"INCLUDE_IN_TOT"
else ifeq ($(AD_MODE),21)
  CPP_DER2:=-D"INCLUDE_IN_TOR"
endif
ifeq ($(ad1),b)
  CPP_ADJ:=-D"INCLUDE_IN_ADJOINT"
endif
ifeq ($(ad1),d)
  CPP_TAN:=-D"INCLUDE_TANGENT"
  ad_tan:=d
else ifeq ($(ad2),d)
  CPP_TAN:=-D"INCLUDE_TANGENT"
  ad_tan:=d
else
  ad_tan:=
endif
```

Figure B.2: Blanking of algorithms from the compiler for enabling the generation of executables with varying capabilities.

Bibliography

[1] Aftosmis, M., Gaitonde, D., and Tavares, T. Behavior of linear reconstruction techniques on unstructured meshes. *AIAA journal 33* (1995), 2038–2049.

[2] Ahlberg, J., and Nilson, E. *The Theory of Splines and Their Applications*, vol. 38. Elsevier, 1967.

[3] Allmaras, S. Analysis of a local matrix preconditioner for the 2-d navier-stokes equations. 358–374.

[4] Ansys. Fluent, CFD simulation software. www.ansys.com.

[5] Ansys. Gambit, Mesh generation for Computational Fluid Dynamics. http://www.ansys.com/.

[6] Ansys. ICEM, Mesh generation software. http://www.ansys.com.

[7] Barth, T. The design and application of upwind schemes on unstructured meshes. *AAIA Paper 89-0366* (1989).

[8] Barth, T. A 3D upwind euler solver for unstructured meshes. *AAIA Paper 91-1548* (1991).

[9] Barth, T. *Aspects of unstructured grids and finite-volume solvers for the Euler and Navier-Stokes equations*. AGARD R-787, Brussels, Belgium, 1992. Special course on unstructured grid methods for advection dominated flows.

[10] Bartholomew-Biggs, M., Brown, S., Christianson, B., and Dixon, L. Automatic differentiation of algorithms. *Elsevier* (2000).

[11] Beyer, H.-G., and Sendhoff, B. Robust optimization - a comprehensive survey. *Computer Methods in Applied Mechanics and Engineering 196*, 33-34 (2007), 3190 – 3218.

[12] Blazek, J. *Computational Fluid Dynamics: Principles and applications*. Elsevier, Oxford, United Kingdom, 2005.

[13] Blender. Free open source 3D content creation suite. http://www.blender.org/.

[14] Bonnans, J. F., Gilbert, J. C., Lemarechal, C., and Sagastizabal, C. A. *Numerical optimization: Theoretical and practical aspects*. Springer-Verlag, 2006.

[15] Brandt, A. Multi-level adaptive computations in fluid dynamics. 100–108. Technical Report AIAA-79-1455, AIAA, Williamsburg, VA.

[16] Broyden, C. The convergence of single-rank quasi-newton methods. *American Mathematical Society 24* (1970).

[17] Cauchy, A. *Methode genrale pour la resolution des systemes d'equations simultanees*. 1847.

[18] Chapman, S. J. Fortran 90/95 for scientists and engineers.

[19] Christakopoulos, F., Jones, D., and Müller, J.-D. Pseudo-timestepping and verification for automatic differentiation derived CFD codes. *Computers & Fluids 46*, 1 (2011), 174 – 179. 10th ICFD Conference Series on Numerical Methods for Fluid Dynamics (ICFD 2010).

[20] Christianson, B. Reverse aumulation and implicit functions. *Optimization Methods and Software 9*, 4 (1998), 307–322.

[21] Courant, R., Friedrichs, K., and Lewy, H. über die partiellen differenzengleichungen der mathematischen physik. *Mathematische Annalen 100* (1928), 32–74. 10.1007/BF01448839.

[22] Courty, F., Dervieux, A., Koobus, B., and Hascoët, L. Reverse automatic differentiation for optimum design: from adjoint state assembly to gradient computation. *Optimization Methods and Software 18*, 5 (2003), 615–627.

[23] Cusdin, P. A. *Automatic sensitivity code for Computational Fluid Dynamics*. 2005. PhD thesis, School of Aeronautical Engineering, Faculty of Engineering, Queen's University, Belfast.

[24] Cusdin, P. A., and Müller, J.-D. Generating efficient code with automatic differentiation. Jyvaskyla, Finland, 24-28 July.

[25] Cusdin, P. A., and Müller, J.-D. On the performance of discrete adjoint cfd codes using automatic differentiation. *International Journal of Numerical Methods in Fluids* (2004).

[26] Dasault. CATIA - the digital product experience. http://www.3ds.com/products/catia.

[27] Davis, L., and Mitchell, M. Handbook of genetic algorithms. *Van Nostrand Reinhold* (1991).

[28] De Boor, C. *A Practical Guide to Splines*. Springer-Verlag, 1978.

[29] Dennis, J. E., J., and More, J. J. Quasi-newton methods, motivation and theory. *SIAM Review 19*, 1 (1977), pp. 46–89.

[30] Drikakis, D., Iliev, O., and Vassileva, D. A nonlinear multigrid method for the three-dimensional incompressible navier-stokes equations. *Journal of Computational Physics 146*, 1 (1998), 301 – 321.

[31] Fan, J., and Yao, Q. *Spline methods*. Springer, 2005.

[32] Fletcher, R. *Practical methods of optimization*. John Wiley & Sons, 1987.

[33] Fogel, D. B. *Evolutionary computation: toward a new philosophy of machine intelligence*. IEEE Press, 1998.

[34] Frink, T., Parikh, P., and Pirzadeh, S. A fast upwind solver for the euler equations on three dimensional unstructured meshes. *AAIA Paper 91 - 0102* (1991).

[35] Gerald, F. *Curves and surfaces for computer-aided geometric design*. Elsevier, 1997.

[36] Geuzaine, C., and Remacle, J.-F. 2D/3D finite element grid generator with a build-in CAD engine and post-processor. http://geuz.org/gmsh/.

[37] Ghate, D., and Giles, M. Efficient hessian calculation using automatic differentiation. *AIAA paper 2007-4056* (2007).

[38] Giles, M. B., Duta, M. C., Müller, J.-D., and Pierce, N. A. Algorithm developments for discrete adjoint methods. *AIAA Journal 41* (2003), 198–205.

[39] GNU. The c preprocessor. http://gcc.gnu.org/onlinedocs/cpp/.

[40] GNU. Gfortran, the gnu fortran project, compiler for gcc, the gnu compiler collection. http://gcc.gnu.org/wiki/GFortran.

[41] GNU. Gnu compiler collection. http://gcc.gnu.org/.

[42] GNU. M4–implementation of the traditional unix macro processor. http://www.gnu.org/software/m4/.

[43] GNU. sed–a stream editor. http://www.gnu.org/software/sed/manual/sed.html.

[44] Godunov, S. A difference scheme for numerical computation of discontinuous solution of hydrodynamic equations. *Math. Sbornik (in Russian) 47* (1959), 271–306.

[45] Goldberg, D. E. *Genetic Algorithms in Search, Optimization, and Machine Learning*, 1 ed. Addison-Wesley Professional, 1989.

[46] Griewank, A., and Walther, A. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, second ed. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2008.

[47] Hascoët, L., and Pascual, V. Tapenade 2.1 user's guide.

[48] Hazra, S., Schulz, V., Brezillon, J., and N.R., G. Aerodynamic shape optimization using simultaneous pseudo-timestepping. *JCP 204* (2005), 46–64.

[49] Hestenes, M. R., and Stiefel, E. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards 49* (1952).

[50] Hildebrand, F. B. Finite-difference equations and simulations.

[51] Jameson, A. Aerodynamic design via control theory. *Journal of Scientific Computing 3* (1988-09-01), 233–260.

[52] Jameson, A., Martinelli, L., and Pierce, N. Optimum aerodynamic design using the navierstokes equations1. *Theoret. Comput. Fluid Dynamics* (1998).

[53] Jameson, A., Schmidt, W., and Turkel, E. Numerical solution of the euler equations by finite volume methods using runge kutta time stepping schemes. *AIAA paper 81-1259* (1981).

[54] Jameson, A., and Vassberg, J. Studies of alternative numerical optimization methods applied to the brachistochrone problem. *Computational Fluid Dynamics Journal 9*, 3 (2000).

[55] Jaworski, A., Cusdin, P., and Müller, J.-D. Uniformly converging simultaneous time-stepping methods for optimal design.

[56] Jaworski, A., and Müller, J.-D. Towards modular multigrid design optimisation. 281–292.

[57] Karush, W. Minima of functions of several variables with inequalities as side constraints. Master's thesis, Dept. of Mathematics, Univ. of Chicago, 1939.

[58] Kuhn, H., and Tucker, A. Nonlinear programming.

[59] Lee, E. *A Simplified B-Spline Computation Routine*, vol. 29. Springer-Verlag, 1982.

[60] Lee, E. *Comments on some B-spline algorithms*, vol. 36. Springer-Verlag, 1986.

[61] Liou, M.-S. On a new class of flux splittings.

[62] Liou, M.-S. A sequel to ausm: Ausm+. *Journal of Computational Physics 129*, 2 (1996), 364 – 382.

[63] Liou, M.-S. A sequel to ausm, part ii: Ausm+-up for all speeds. *Journal of Computational Physics 214*, 1 (2006), 137 – 170.

[64] Liou, M.-S., and Steffen, C. J. J. A new flux splitting scheme. *Journal of Computational Physics 107*, 1 (1993), 23 – 39.

[65] L.L., S., Taylor, A., Green, L., Newman, P., Hou, G., and Korivi, V. First- and second-order aerodynamic sensitivity derivatives via automatic differentiation with incremental iterative methods. *Journal of Computational Physics 129*, 2 (1996), 307 – 331.

[66] Martinelli, L. *Calculations of viscous flows with a multigrid method*. Princeton University, 1987. PhD Thesis, Department of Mechanical and Aerospace Enginnering.

[67] Martinelli, M., and Duvigneau, R. On the use of second-order derivatives and metamodel-based monte-carlo for uncertainty estimation in aerodynamics. *Computers & Fluids 39*, 6 (2010), 953 – 964.

[68] Martinelli, M., and Hascoet, L. Tangent-on-tangent vs tangent-on-reverse for second differentiation of constrained functionals.

[69] Martins, J., Sturdza, P., and Alonso, J. J. The complex-step derivative approximation. *ACM Trans. Math. Softw. 29*, 3 (2003), 245–262.

[70] Mavriplis, D. A discrete adjoint–based approach for optimisation problems on three–dimensional unstructured meshes. *AIAA Journal 45*, 4 (2007), 740–759.

[71] Mavriplis, D., and Jameson, A. Multigrid solution of the Navier-Stokes equations on triangular meshes. *AIAA journal* (1990), 1415–1425.

[72] Mavriplis, D., Jameson, A., and Martinelli, L. Multigrid solution of the navier-stokes equation on triangular meshes. *AAIA-89-0120* (1989), 1–10.

[73] Michalewicz, Z. *Genetic Algorithms Plus Data Structures Equals Evolution Programs*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1994.

[74] Mohammadi, B., and Pironneau, O. Mesh adaption and automatic differentiation in a cad-free framework for optimal shape design. *International Journal for Numerical Methods in Fluids 30* (1999), 127–136.

[75] Mulas, M., Chibbaro, S., Delussu, G., Di Piazza, I., and Talice, M. Efficient parallel computations of flows of arbitrary fluids for all regimes of reynolds, mach and grashof numbers. *International Journal of Numerical Methods for Heat & Fluid Flow 12* (2002), 637–657.

[76] Müller, J.-D. delaundo, "Two–dimensional triangulation based on the frontal delaunay method (frod)". http://www.cerfacs.fr/ muller/delaundo.html.

[77] Müller, J.-D. HiP, "Multi-function grid-processor". http://www.cerfacs.fr/ muller/hip.html.

[78] Müller, J.-D. ipol, "Point distribution on analytic or discrete curves". http://www.cerfacs.fr/ muller/delaundo.html.

[79] Müller, J.-D., Moinier, P., and Giles, M. B. Edge-based multigrid and preconditioning for hybrid grids. *AIAA Paper* (1999), 99–3339.

[80] NASA. FUN3D analysis and design - nasa vertex–centred solver. http://fun3d.larc.nasa.gov/.

[81] NASA. Nparc alliance validation archive. www.grc.nasa.gov/.../m6wing.html.

[82] NASA. Turbulent benchmarck results on a flat plate using the spalart - allmaras turbulence model. http://turbmodels.larc.nasa.gov/flatplate_sa.html.

[83] Naumann, U., Maier, M., Riehme, J., and Christianson, B. Automatic first- and second-order adjoints for truncated newton. *In M. Ganzha et al., editor,*

Proceedings of IMCSIT07: Workshop on Computer Aspects of Numerical Algorithms (CANA07) (2007), 541555.

[84] Ohtakea, Y., Belyaevb, A., and Bogaevskic, I. Mesh regularization and adaptive smoothing. *Computer-Aided Design 33*, 11 (2001), 789 – 800.

[85] Othmer, C. A continuous adjoint formulation for the computation of topological and surfacesensitivities of ducted flows. *International Journal for Numerical Methods in Fluids 58*, 8 (1998), 861877.

[86] Papadimitriou, D., and Giannakoglou, K. A continuous adjoint method with objective function derivatives based on boundary integrals, for inviscid and viscous flows. *Computers & Fluids 36*, 2 (2007), 325 – 341.

[87] Papadimitriou, D., and Giannakoglou, K. Computation of the hessian matrix in aerodynamic inverse design using continuous adjoint formulations. *Computers & Fluids 37*, 8 (2008), 1029 – 1039.

[88] Patankar, S. V. *Numerical heat transfer and fluid flow*. 1980. Series in computational methods in mechanics and thermal sciences.

[89] Piegl, A. Tiller, W. *The NURBS book*. Springer, 1997.

[90] Pierce, N., and Giles, M. Preconditioned multigrid methods for compressible flow calculations on stretched meshes.

[91] Pironneau, O. On optimum design in fluid mechanics. *Journal of Fluid Mechanics 64*, 01 (1974), 97–110.

[92] Putko, M., Newman, P., Taylor, A., and Green, L. Approach for uncertainty propagation and robust design in cfd using sensitivity derivatives. *AIAA journal* (2001).

[93] Rampfkeil, M., and Mavriplis, D. Efficient hessian calculations using automatic differentiation and the adjoint method with applications. *AIAA 48*, 10 (2010), 2406 – 2417.

[94] Richtmeyer, D., and Morton, K. *Difference Methods for Initial Value Problems*. 1967. 2nd edition.

[95] Roe, P. Upwind differencing schemes for hyperbolic conservation laws with source terms. In *Nonlinear Hyperbolic Problems*, C. Carasso, D. Serre, and P.-A. Raviart,

Eds., vol. 1270 of *Lecture Notes in Mathematics*. Springer Berlin / Heidelberg, 1987, pp. 41–51. 10.1007/BFb0078316.

[96] Roe, P. Approximate riemann solvers, parameter vectors, and difference schemes. *Journal of Computational Physics 135* (August 1997), 250–258(9).

[97] Schmidt, S., Ilic, C., Gauger, N., and Schulz, V. Shape gradients and their smoothness for practical aerodynamic design optimization. *Universität Trier* (2008). Preprint-Number SPP1253-10-03.

[98] Snyman, A. J. *Practical Mathematical Optimization: An Introduction to Basic Optimization Theory and Classical and New Gradient-Based Algorithms.* 2005.

[99] Squire, W., and Trapp, G. Using complex variables to estimate derivatives of real functions. *SIAM Review 40*, 1 (1998), 110–112.

[100] Trefethen, L., and Bau, D. *Numerical Linear Algebra.* 1997.

[101] Tropics, P. Tapenade. http://tapenade.inria.fr:8080/tapenade/ index.jsp, INRIA Sophia-Antipolis, France.

[102] Van Leer, B. Towards the ultimate conservative difference scheme v. a second-order sequel to godunov's method. *Journal of Computational Physics Volume 135*, Number 2 (1997), 229 – 248.

[103] Van Leer, B., Tai, C.-H., and Powell, K. Design of optimally smoothing multistage schemes for the euler equations. *AIAA paper 89-1933* (1989).

[104] Venkatakrishnan, V. On the accuracy of limiters and convergence to steady state solutions. *AAIA Paper 93-0880* (1993).

[105] Venkatakrishnan, V. Convergence to steady state solutions of the euler equations on unstructured grids with limiters. *Journal of Computational Physics 118* (April 1995), 120–130(0).

[106] Webster, R. Algebraic multigrid and incompressible fluid flow. *International Journal for Numerical Methods in Fluids 53*, 4 (2007), 669–690.

[107] Wesseling, P. *An Introduction to Multigrid Methods.* 1992.

[108] Whitfield, D., and Janus, J. Three–dimensional unsteady euler equations solution using flux vector splitting. *AIAA paper 84-152* (1984).

[109] Xu, S. Cad-based aerodynamic shape-optimisation with adjoint method, 9 month progress report. *Queen Mary, University of London* (2011).

[110] Zervogiannis, T., Papadimitriou, D., and Giannakoglou, K. Total pressure losses minimization in turbomachinery cascades using the exact hessian. *Computer Methods in Applied Mechanics and Engineering 199*, 41 - 44 (2010), 2697 – 2708.

[111] Zhu, C., Byrd, R., Nocedal, J., and Morales, J. L. L-bfgs-b, software for large-scale bound-constrained optimization. http://users.eecs.northwestern.edu/ nocedal/lbfgsb.html.